[redacted]

0.84
l6r
0.527 Report No. UIUCDCS-R-72-527 Math
cop.2

A MULTIPROCESSOR FOR SIMULATION APPLICATIONS

by

Edward Willmore Davis, Jr.

June 1972

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. UIUCDCS-R-72-527

A MULTIPROCESSOR FOR SIMULATION APPLICATIONS*

by

Edward Willmore Davis, Jr.

June 1972

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

*This work was supported in part by the National Science Foundation under Grant No. US NSF GJ 27446 and was submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science, June 1972.

A MULTIPROCESSOR FOR SIMULATION APPLICATIONS

Edward Willmore Davis, Jr., Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1972

Multiprocessor systems have generally been designed for applications with arrays of data which can be operated on in parallel. In this thesis an application area which does not contain such readily identifiable parallelism is examined. Discrete time simulation is found to contain several distinct levels at which potential for concurrent execution exists. The levels are used to guide the organization of a multiprocessor designed for simulation applications.

Both software and hardware aspects of the problem are covered. Features of the system include a special processor used to evaluate conditional jump trees; clusters of simple, fixed point arithmetic processors; a unit to form and dispatch tasks to the processors; and a memory system which includes a read only program memory.

ACKNOWLEDGMENT

The author wishes to express sincere gratitude to his advisor, Professor David J. Kuck. The many hours of discussion, suggestions, advice, and encouragement were fundamental to the development and completion of this dissertation.

The team working on the Fortran program analysis system deserve thanks. In particular, R. Towle did an excellent job maintaining the system and revising it for the needs of the author, and R. Strebendt provided instruction on system use and interpretation of the results.

The dedicated effort of Mrs. Vivian Alsip in typing the final manuscript, as well as much of the draft copy, is gratefully acknowledged. The typing done by Mrs. Diana Mercer also deserves credit. Fred Hancock, of the Center for Advanced Computation, voluntarily drew several figures when the Department of Computer Science drafting group became overloaded. The support of the research by the Department of Computer Science and the Center for Advanced Computation is greatly appreciated.

Finally, the author wishes to thank his wife, Jo Ann, for her patience and help in this effort.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

1.1 Overview

Speeding up the execution of programs by means of compile time algorithms and machine organization is the general topic of this thesis. The approach taken is to study an application area, then design a machine to take advantage of characteristics of programs for the application, parallelism in the problem, and processing requirements. This study leads to a multiprocessor configuration with a hierarchy of processors and memories.

Discrete time simulation is the application area selected. Simulation languages, particularly the General Purpose Simulation System [7], GPSS, are examined.

The purpose of this study is to achieve speedup through machine organization with the use of available logic and memory devices. The speedup does not come from faster execution of individual instructions. Instead, the approach taken is to have more than one instruction in execution simultaneously. GPSS is studied to detect parallelism and provide guidelines for the design of a multiprocessor machine.

Simulation programming does not include operations on arrays of data, the feature most commonly associated with machines having more than, say, two processors. An early observation on simulation program characteristics was that conditional jump statements occur with great frequency. To significantly speed up the execution it is necessary to do better than serial processing of conditional jumps. In this thesis algorithms and a special hardware unit are designed for processing trees of conditional jumps.

The thesis is experimental in nature. It includes the analysis, for execution on a multiprocessor, of Fortran programs with approximately 1000 statements. Several GPSS programs are individually analyzed for prospects of concurrency in execution. A simulation system is used to test the performance of the proposed machine organization in the execution of GPSS programs.

1.2 Thesis Organization

This thesis is organized as three chapters which present the details of the problems and solutions, a final chapter which summarizes results, and an appendix which describes a software test system for generating and verifying some of the results.

Chapter 2 is concerned with the problem of speeding up the execution of programs with many conditional jumps. Software algorithms to increase the execution concurrency are presented. The algorithms modify the original program, without changing the logic, such that better use of the multiprocessor system is made. A hardware unit for evaluating conditional jump statement trees is presented. This "decision processor" operates in conjunction with the arithmetic processors to select a path through trees of many levels.

Chapter 3 discusses discrete time simulation languages and examines GPSS in some detail. Parallelism in GPSS and the potential for concurrent execution are the major topics.

A multiprocessor machine organization for languages like GPSS is designed in Chapter 4. The machine consists of several clusters of processors, a memory system matched to the processor requirements, and a unit to coordinate the processor clusters in their execution of a program. Processors within a cluster have a common control unit; however, the instruction stream to each processor can differ. Clusters operate independently from each other and are individually capable of executing a complete program.

Conclusions are presented in Chapter 5. The Appendix is concerned with a system for simulating the machine organization of Chapter 4.

2. CONCURRENT PROCESSING OF CONDITIONAL JUMP STATEMENTS

2.1 Introduction

The purpose of a multiprocessor machine organization is to speed up program execution. Speedup is achieved by using the parallelism or concurrency that can be extracted from a program to keep more than one processor busy. For machines such as ILLIAC IV [9,15], STAR [3], or the array associative processor STARAN [5], the parallelism is largely due to operations on arrays of data. Other machine organizations or parallelism extraction schemes use tree height reduction DO loop and recurrence relation expansion, back substitution, and independent blocks of assignment statements to get a more general form of parallelism [10]. These schemes may add redundant operations or increase the number of useful operations but they do reduce the number of steps required for execution.

All of the above techniques work on sections of code that exist between the statements that control the flow of program execution. When a conditional jump (an IF statement) occurs there is a reversion to serial execution. For programs with very few conditional jumps this is not a serious weakness. For programs or parts of programs with a high ratio of IF to assignment statements, this serial execution can degrade significantly the efficiency of a multiprocessor.

In this chapter algorithms and hardware for speeding up the execution of programs with many IF statements are examined. The hardware is a special purpose processor designed into the multiprocessor system of Chapter 4. Terms related to the algorithms and hardware are defined in section 2.2. Compile time preparation of programs to use the processor is introduced in 2.3.

Section 2.4 covers the processor and 2.5 discusses its operation.

2.2 Decision Trees in Programs and Processors

Decision statements in programs are those that determine the next instruction to be executed from two or more possible choices. These statements typically begin with the conjunction "IF". The IF statements considered here are the logical type where a boolean variable is the basis of choice between two next instructions. Other types of IFs can be converted to the two way jump form.

When at least one of the instructions selected by an IF is also an IF, a tree of IF statements called a *decision tree* is formed. A single IF in the tree is a *node*. Establish the convention that the branch taken when the boolean variable is true is pictured as leaving the node to the right. Then the decision tree corresponding to the code:

```
IF (A) THEN IF (B) THEN W;
                    ELSE X;
       ELSE IF (C) THEN Y;
                    ELSE Z;
```

can be drawn as in Figure 2.1.

An *exit* from the tree occurs when a statement other than an IF is next. In Figure 2.1, W, X, Y, and Z represent exits. The directed line segments between nodes are *branches*. Any sequence of branches followed to reach an exit is a *path* through the tree. The path taken on a given execution of the tree identifies the exit and is the *result* for that execution. A *single input node* is a node with only one branch into it. The discussion and examples in this chapter, with the exception of section 2.5.2.2, assume all nodes are single input nodes.

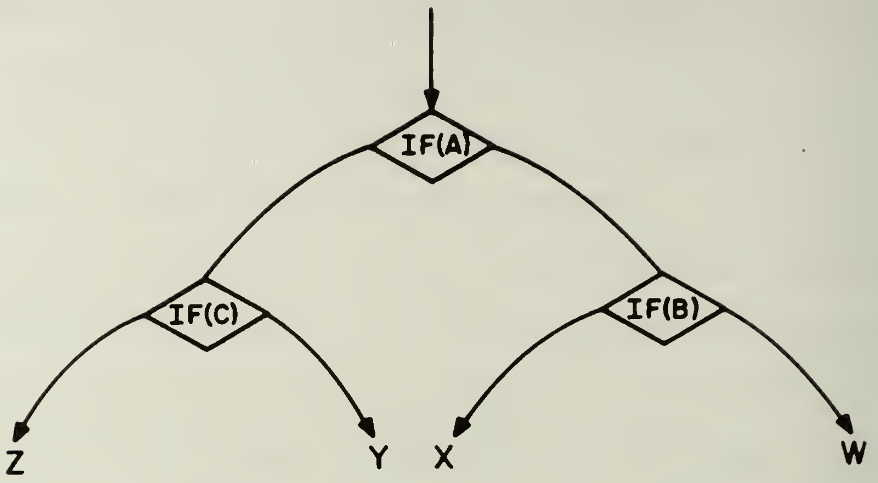


Figure 2.1. A Decision Tree

Elements of the decision tree are labeled according to the rules below. Figure 2.2 shows the naming scheme.

1. The root node is λ.
2. The name of a branch directed out of a node is the name of the node concatenated with 1 or 0 according to whether the branch leaves the node to the right or left. For concatenation, λ is a null element.
3. Nodes other than λ are given the name of the branch directed into them.
4. Paths and exits are identified by the name of the branch which is the exit.

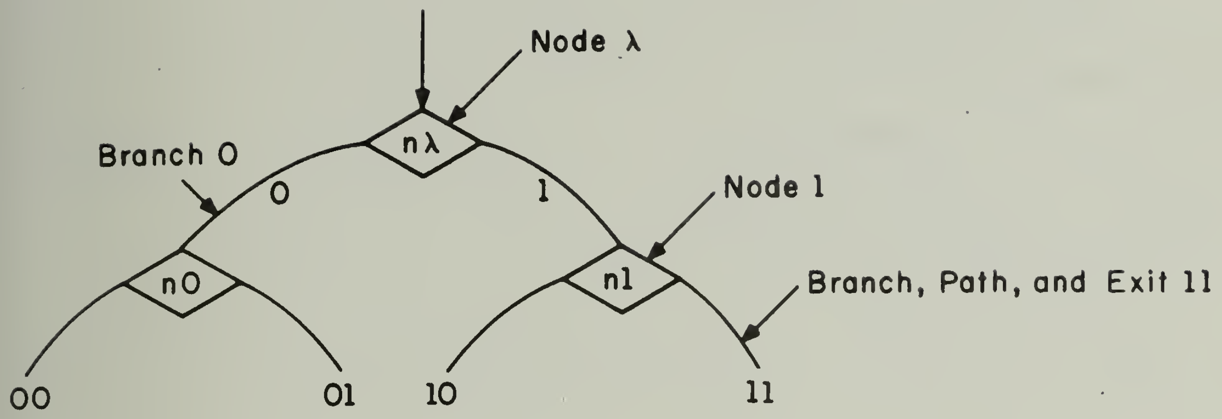


Figure 2.2. Decision Tree Labels

The nodes on a given <u>level</u> of the tree are the set of nodes which have the same number of bits in their name. Levels are numbered sequentially beginning with one at the root such that at level i there are 2^{i-1} possible nodes. Let ℓ be the number of levels in the tree. A tree is <u>full</u> if all of the $2^{\ell}-1$ possible nodes are present. There are 2^{ℓ} exits from a full tree. In an informal way "length" will refer to the number of levels and "shape" will refer to the number and distribution of nodes in a tree.

Programs in general have assignment statements interspersed with decision statements. Those parts of a program where the ratio of decision statements to assignment statement operations is larger than an experimentally determined threshold are called <u>IF trees</u>. When the assignment statements are removed from an IF tree a decision tree is formed. An algorithm is presented in section 2.3.3 for movement of assignment statements out of an IF tree such that correct execution of the program is not disturbed. The algorithm allows the formation of larger decision trees than exist naturally in a program.

Now consider a processor to evaluate decision trees in a parallel way. The length and shape of a programmed tree can vary, limited only by the syntax of the language being used. The length and shape of processing equipment is fixed by hardware design. The fixed hardware must be capable of processing trees of any length or shape. Let k be the number of levels in the decision processor. The processor is designed for a full tree so 2^k-1 nodes can be evaluated. Longer full trees require repeated use of the processor.

Hardware nodes are numbered from left to right within levels and from level one to level k. These numbers are the decimal equivalent of decision tree binary node numbers with a leading one attached. The tree that descends from each node is a <u>sector</u> which is given the name of the sector root node. Sector 1 includes the complete processor tree. All other sectors represent sub-trees. Control is provided in the processor to select a particular sector for evaluation, providing isolation and independence from other sectors.

A labeled two level processor tree is shown in Figure 2.3.

The mapping of $j \leq k$ levels of decision tree nodes into the k levels of processor tree nodes is a one-one mapping. It is not in general "onto" since the decision tree shape may differ from the fixed processor tree. A processor node which corresponds to a decision tree node is a <u>decision node</u>.

All processor exits are from level k. Any decision tree exit from a node which does not map onto level k must "use" a node at each succeeding

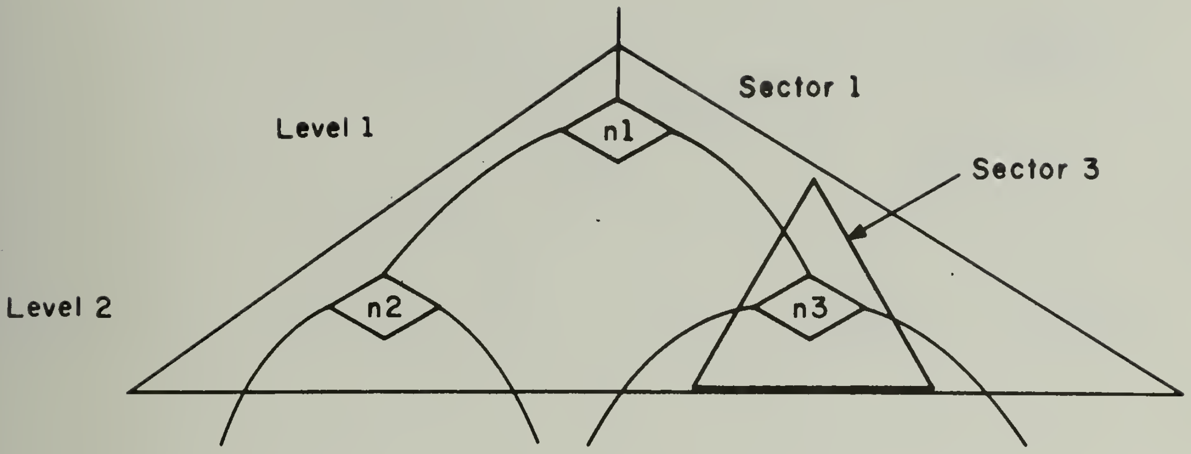


Figure 2.3. Processor Tree

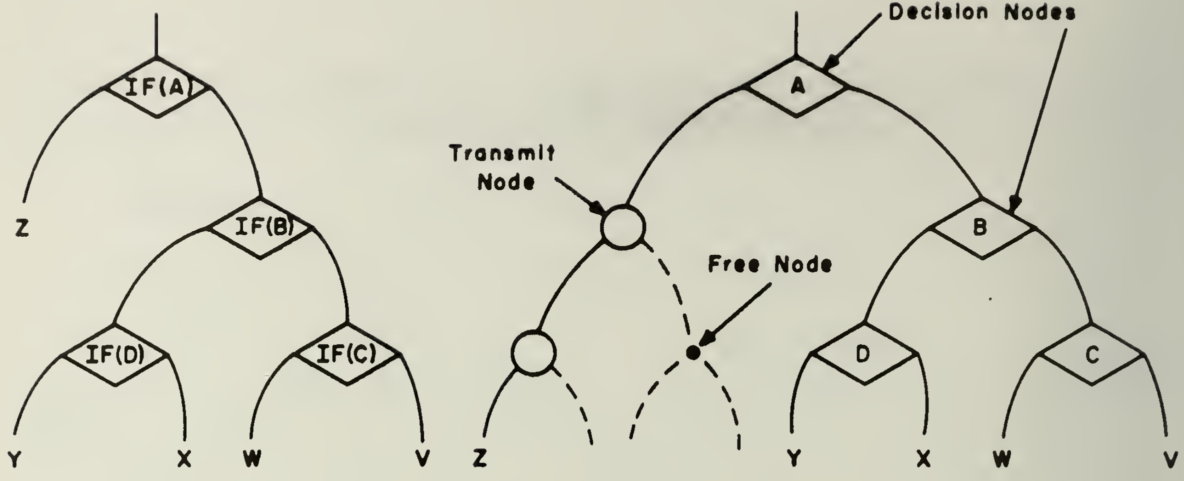
level down to k. Nodes that are used to transmit the output of a decision node to an exit, but which do not take part in the decision making process, are transmit nodes. Transmit nodes are assigned a logical value of zero.

There is a third class of nodes in the processor tree. A free node is one which is neither a decision nor a transmit node.

In drawings to follow, the symbol "◇" will represent a decision node, "○" will represent a transmit node, and "•" a free node.

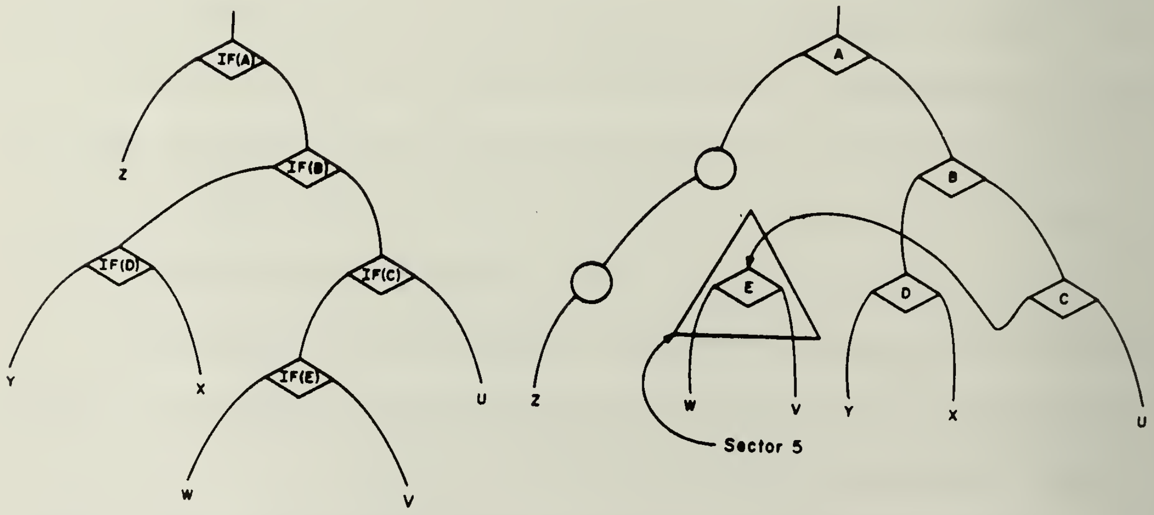
As an example of several of the above definitions consider the three level trees in Figure 2.4. The exits from the free node will never be on a selected path since the free node is on the "1" output branch of a transmit node which always has the output "0".

Suppose the decision tree had an additional node on level four. That node can be mapped into the free node if the tree processing is controlled as follows. Evaluate sector one; the complete three level processor tree.



(a) Decision Tree (b) Corresponding Processor Tree

Figure 2.4. Node Classification



(a) Decision Tree (b) Corresponding Processor Tree

Figure 2.5. Use of a Free Node

If the exit is the branch that leads to the level four node, evaluate sector five to get the final exit. The situation is illustrated in Figure 2.5.

The mapping of a sub-tree of a decision tree node into a higher level sector in the processor tree is called <u>folding</u>. The process of evaluating a sector is a <u>cycle</u>. If folding is used to fill free nodes, evaluation may take as many cycles as there are folds, plus one for sector one.

If not all decision tree nodes can be mapped into the processor the tree is evaluated by repeated use of the processor. Each use related to a given tree is an <u>iteration</u>. An iteration may involve many cycles.

At this point much of the terminology has been explained and the execution scheme introduced. Now consider the software required for this processor.

2.3 Software Aspects of IF Trees and Decision Trees

Locating IF trees in programs is based on an algorithm described in [2]. The algorithm forms a trace of all paths through a program. Detection of an IF statement on a path activates a counter of assignment statement operations. As long as the operation count is below a specified threshold, another IF statement on the path is classified as being in a tree with the predecessor. The counter is reset and operation counting begins again.

When the operation count exceeds the threshold, an exit from the IF tree has been found. The occurrence of input or output statements or subroutine calls also mark an exit from the tree.

Examination of several programs has shown that a threshold value near 10 gives trees which can be processed reasonably well using the techniques of this chapter. Assuming an IF tree has been located, the preparation for processing that must be done at compile time is described in this section.

2.3.1 Overall Processing Scheme

Three actions are involved in the compile time preparation of IF trees for concurrent evaluation. They are mentioned here, then described in more detail in following sections. First, each relational expression which is part of the argument of an IF statement is converted to an assignment statement the left hand side (lhs) of which is a logical variable. The logical variable replaces the relational expression in the argument. The second item is the movement of assignment statements to a position ahead of the remaining decision tree. Third is the mapping of nodes from the decision tree to the processor.

Four steps are required for execution. In the first step assignment statements are evaluated in parallel. The second step is determination of the boolean value of logical arguments. Third, evaluation of the tree in the processor. Fourth, selection of assignments from step one that were on the execution path.

2.3.2 Arguments of IF Statements

An IF statement argument may contain a boolean variable, a relational expression (rel ex), or a logical expression (log ex). For input to the processor tree the argument must be a boolean value. This section discusses the handling of arguments to arrive at the boolean value.

2.3.2.1 Relational Expressions

Relational expressions can be converted to assignment statements (assign sts) which yield the correct boolean value upon examination of the sign or magnitude of the result. As an example, "$X \geq Y$" can be converted to "$B = X-Y$" where the sign of B is a boolean variable indicating the result.

Algorithm 2.1 converts a rel ex to an assign st. The algorithm is given for a machine where the smallest quantity that can be added is ϵ. For an integer machine $\epsilon = 1$. Each assign st formed creates a new logical variable in the program which must be given a unique name. The names generated by the algorithm are the concatenation of a name which does not otherwise exist in the program and the state of a counter.

Let $S(X)$ be a boolean variable representing the sign of X with "+" = 1 and "-" = 0. The sign of $X = 0$ is "+". Let $M(X)$ be a boolean variable representing the magnitude of X. If the magnitude is zero, $M = 0$. If the magnitude is non-zero, $M = 1$. An overbar represents inversion.

Algorithm 2.1: Relational Expression Conversion

0. On the first use of this algorithm on a program, select an unused variable name, Y, and set $I = 1$. On all uses subsequent to the first, enter at step 1.
1. Use Table 2.1 to change the relational expression given in column one to the corresponding assignment statement in column two. Variable X is Y concatenated with I.

Relational Expression	Assignment Statement	Boolean Variable
$L < R$	$X = R-L-\epsilon$	$S(X)$
$L \leq R$	$X = R-L$	$S(X)$
$L = R$	$X = L-R$	$\overline{M}(X)$

Table 2.1. Relational Expression Conversion

Relational Expression	Assignment Statement	Boolean Variable
$L \neq R$	$X = L-R$	$M(X)$
$L \geq R$	$X = L-R$	$S(X)$
$L > R$	$X = L-R-\epsilon$	$S(X)$

Table 2.1 (continued). Relational Expression Conversion

2. Replace the relational expression in the argument with the boolean variable from the corresponding column three entry.
3. Increment I.

2.3.2.2 Logical Expressions

Decision trees fan-out from a root node to more than one possible next statement. Logical expressions as arguments of IFs do the inverse. Given a log ex of at least two variables, a fan-in tree can be formed to give the boolean value. A decision processor designed for the fan-out case is not well suited for evaluating log exs. This section considers means of treating log exs.

Assume the decision processor has no capability to evaluate log exs. The logical IF can be rewritten equivalently as more than one IF where each new argument is a boolean variable. The logical operator connectives are achieved by the way IFs are connected in the program. The basic AND, OR connectives are shown by examples below. Larger expressions are managed by repeated application of these connections.

(a) AND operator

```
Given:    IF (A·B) THEN T; ELSE F;
Rewrite:  IF A THEN IF B THEN T;
                         ELSE F;
               ELSE F;
```

(b) OR operator

```
Given:    IF (A ∨ B) then T; ELSE F;
Rewrite:  IF A THEN T;
               ELSE IF B THEN T;
                         ELSE F;
```

A recent study of a large number of Fortran programs uncovered very few logical IFs with more than one operator in a log ex. [10]. The study did reveal reasonably frequent use of the one operator argument. It therefore seems practical to provide in the decision processor the capability to accept two operand logical expressions. This will be discussed further in section 2.4.1 on input to the processor.

2.3.3 Assignment Statement Movement

Decision trees of more than a few levels will not appear often in programs. Rather, the more general IF tree, the mixture of IF and assignment statements will be present. This section gives an algorithm for moving assign sts out of an IF tree, leaving a decision tree with possibly more levels and certainly more nodes.

Assignment statements are tagged before moving to identify their position in the IF tree. After movement the statements can be analyzed to determine parallelism and executed in parallel. Statements which may not be on the result path are executed concurrently with those that are, since the path is unknown. Thus the results of the block of assign sts are considered temporary pending determination of the result path.

A typical IF tree is shown in Figure 2.6. Here b and f represent boolean and arithmetic functions.

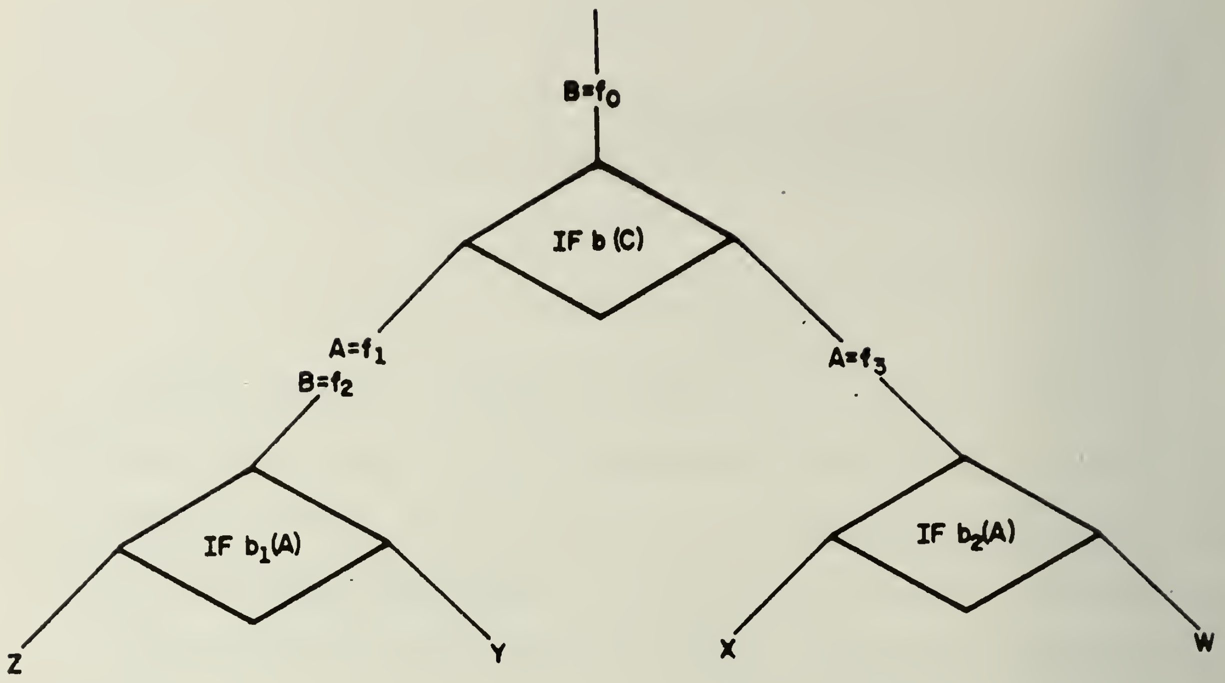


Figure 2.6. An IF Tree

In Figure 2.6, removal of the three assign sts from the IF tree implies a means of distinguishing the two assignments to A and of knowing which of the three were on the result path. The boolean functions of A must be evaluated with the proper value for A. To accomplish this a descriptor is attached to each lhs variable. If such a variable is later read, in the same path through the tree, the same descriptor is attached to that occurrence of the variable also.

The intuitive concept of a predecessor is used in Algorithm 2.2. Some properties of this relation as used here are given. The relation applies to both nodes and branches. "Immediate" is used to mean closest or most direct. The immediate node predecessor of nodes $\alpha 0$ and $\alpha 1$ is node α. A branch is the immediate branch predecessor of the node with the same name. A predecessor of node or branch α is a predecessor of αj, j a binary number.

Algorithm 2.2: Assignment Statement Movement

1. Scan the IF tree from level one to level ℓ applying steps 2, 3, and 4.
2. Attach as a descriptor to the lhs variable of each assignment statement the name of the branch in which the statement occurs. Move the statement to a position above the IF tree.
3. Examine the logical argument of each node and the rhs of each assignment statement, for variables which have been given descriptors in a predecessor branch. Attach the corresponding descriptor to every such variable. If the variable has been given multiple predecessor descriptors, use the most immediate one. Since each higher level contributes one bit to the length of a descriptor assigned at a level, the most recent assignment of a value to a variable is represented by the longest descriptor.
4. Form assignment statements from relational expressions in nodes according to Algorithm 2.1. Move these statements to a position above the IF tree.

At this point node arguments consist of boolean variables or logical expressions and the IF tree has been cleared of assign sts. The IF tree has been converted to a block of assign sts followed by a decision tree. Figure 2.6 can be drawn as in Figure 2.7 after application of Algorithm 2.2.

All assign sts in the block are candidates for execution in parallel. The decision tree can be evaluated in parallel in the decision processor.

Section 2.4 to follow describes the decision processor hardware.

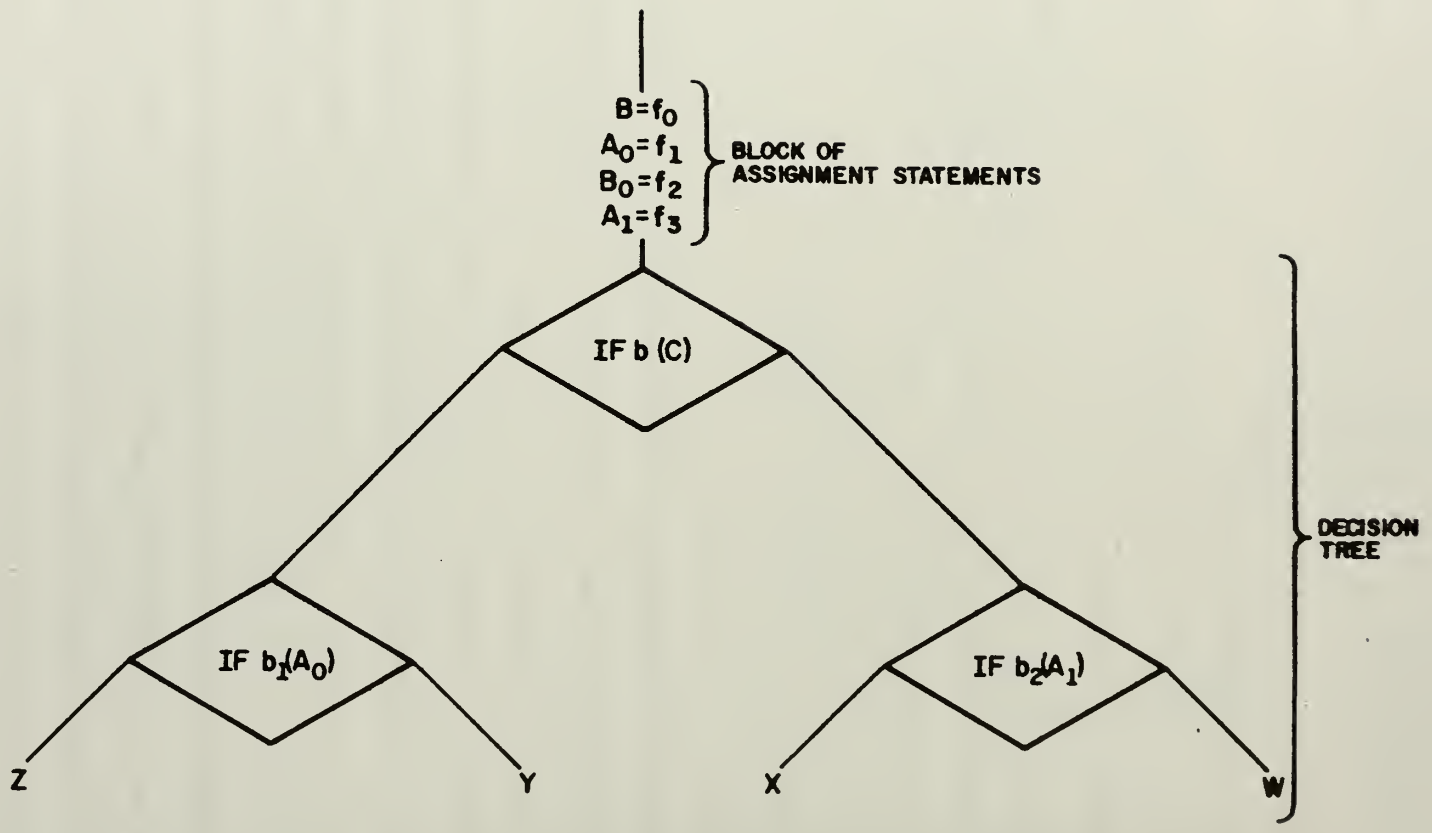


Figure 2.7. The Decision Tree Derived from Figure 2.6

2.4 Decision Tree Processor Hardware

Evaluation of the decision trees defined in the previous sections is carried out on a decision processor. This special purpose processor is designed to operate in conjunction with a multiple arithmetic processor configuration. The function of the decision processor is to accept boolean values corresponding to decision tree nodes and to return information related to the path through the tree. Figure 2.8 is a block diagram of the hardware. The processor has a tree structure with a capacity of 2^k-1 nodes. That is, k is the number of tree levels in the processor.

An input register is used to receive boolean node values from the arithmetic processors. The register has a bit per node of decision processor capacity and a fixed structure relating bits to positions of nodes in trees. A tree decoder identifies the path through the tree. The result of decoding points to the address of the next statement to be executed.

Potential next statement addresses are stored in a small memory which is loaded for each use of the decision processor. For a k level processor tree there are 2^k possible exits. In reality the number of exits programmed is often less than a fourth of that number.

When a programmed tree has more than k levels it is necessary to use more than one evaluation cycle to determine the final result. A register is provided to save the path results on each cycle. When processing of a tree is completed the path register identifies the total path taken and likewise provides a means of identifying the assignment statements which were originally on the path but were moved ahead of the tree.

The design and construction of these components is explained more

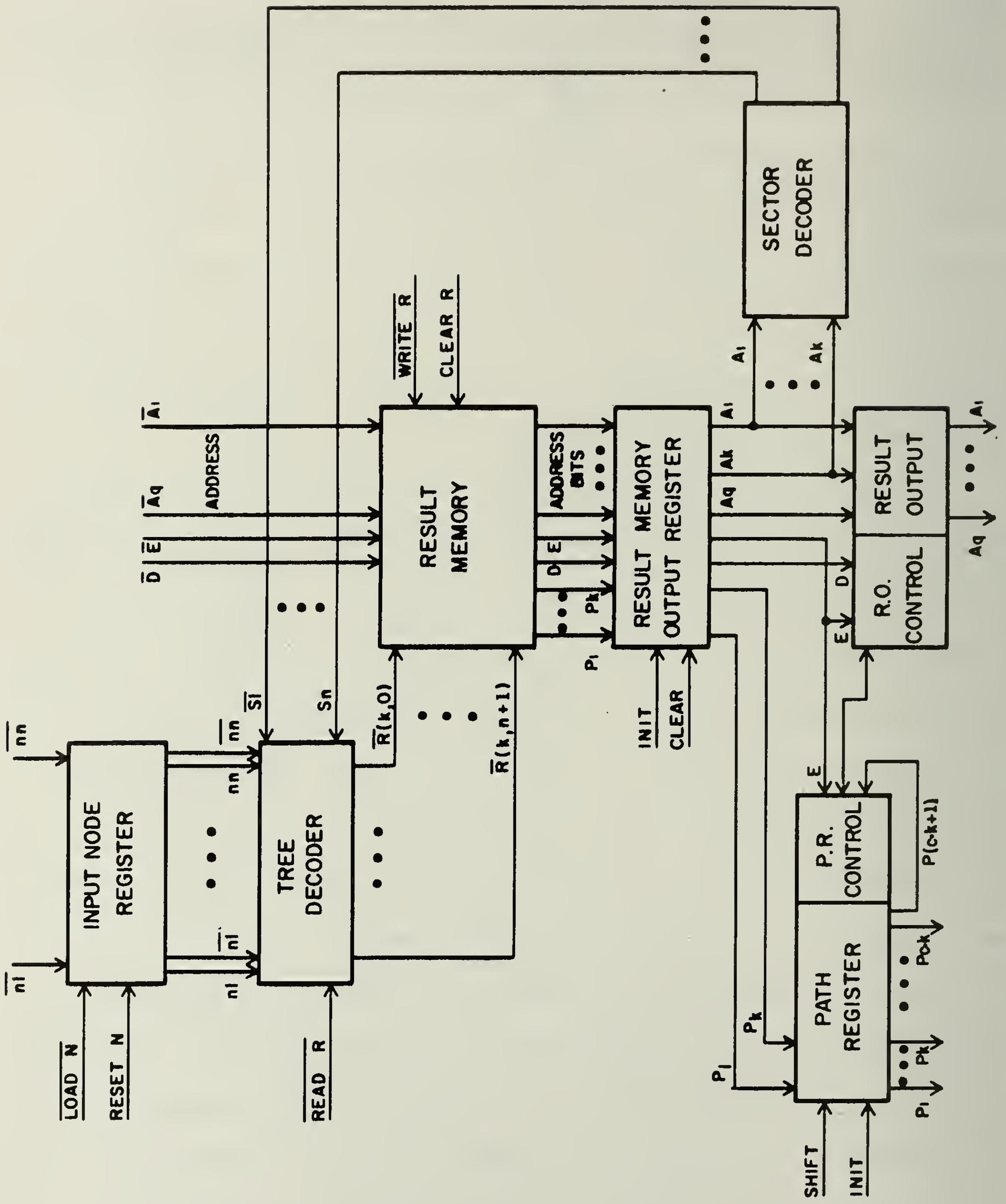


Figure 2.8. Decision Tree Processor

fully in following sections. Optional designs are given for several components. Detailed logic for a three level processor is shown in Figure 2.9. Tables 2.2 and 2.3 contain summary gate counts and timing information.

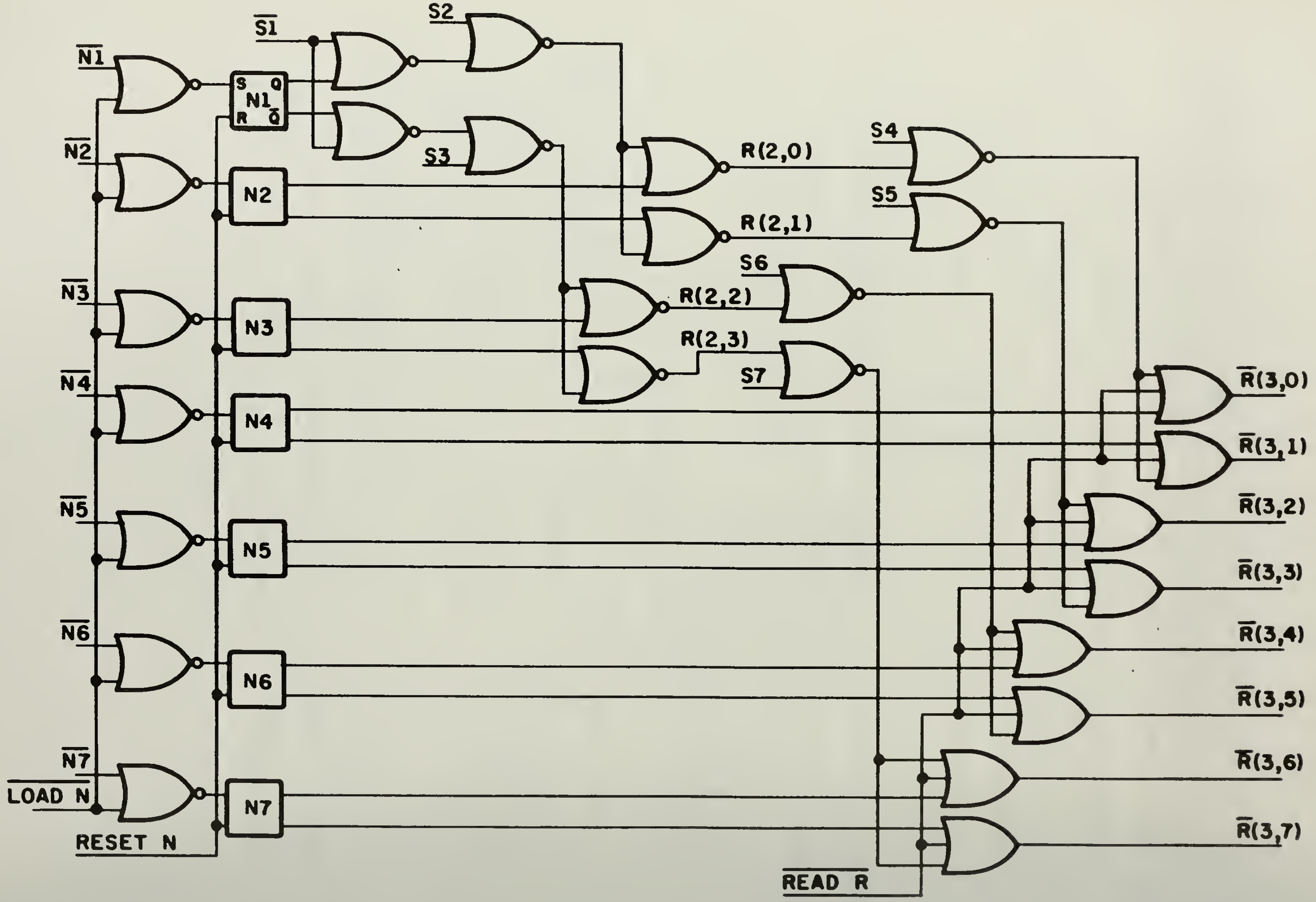
Notation used in drawings and equations is explained here. Individual nodes and sectors are labeled n(i) and S(i) respectively, $1 \leq i \leq (2^k-1)$ following the convention in section 2.2. When a label is used as a signal name it corresponds to a logical one. Let $R(\ell,b)$ be the signal name for branch b at level ℓ in the tree decoder, where b is the decimal equivalent of the binary branch name. For example, R(2,0) is the level 2 branch "00"; R(2,3) is branch "11" at level 2. The output of the k level decoder is $R(k,b)$, $0 \leq b \leq (2^k-1)$.

2.4.1 Input Node Register

Each node in the processor tree is implemented by a specific flip flop, f/f, in this register. The register is reset initially. Data is gated into the set inputs of each f/f such that a f/f is set when the boolean value for that node is one. The f/f remains reset if either the boolean value is zero or the node is not a decision node. The logic for a three level node register is shown in Figure 2.9 (a).

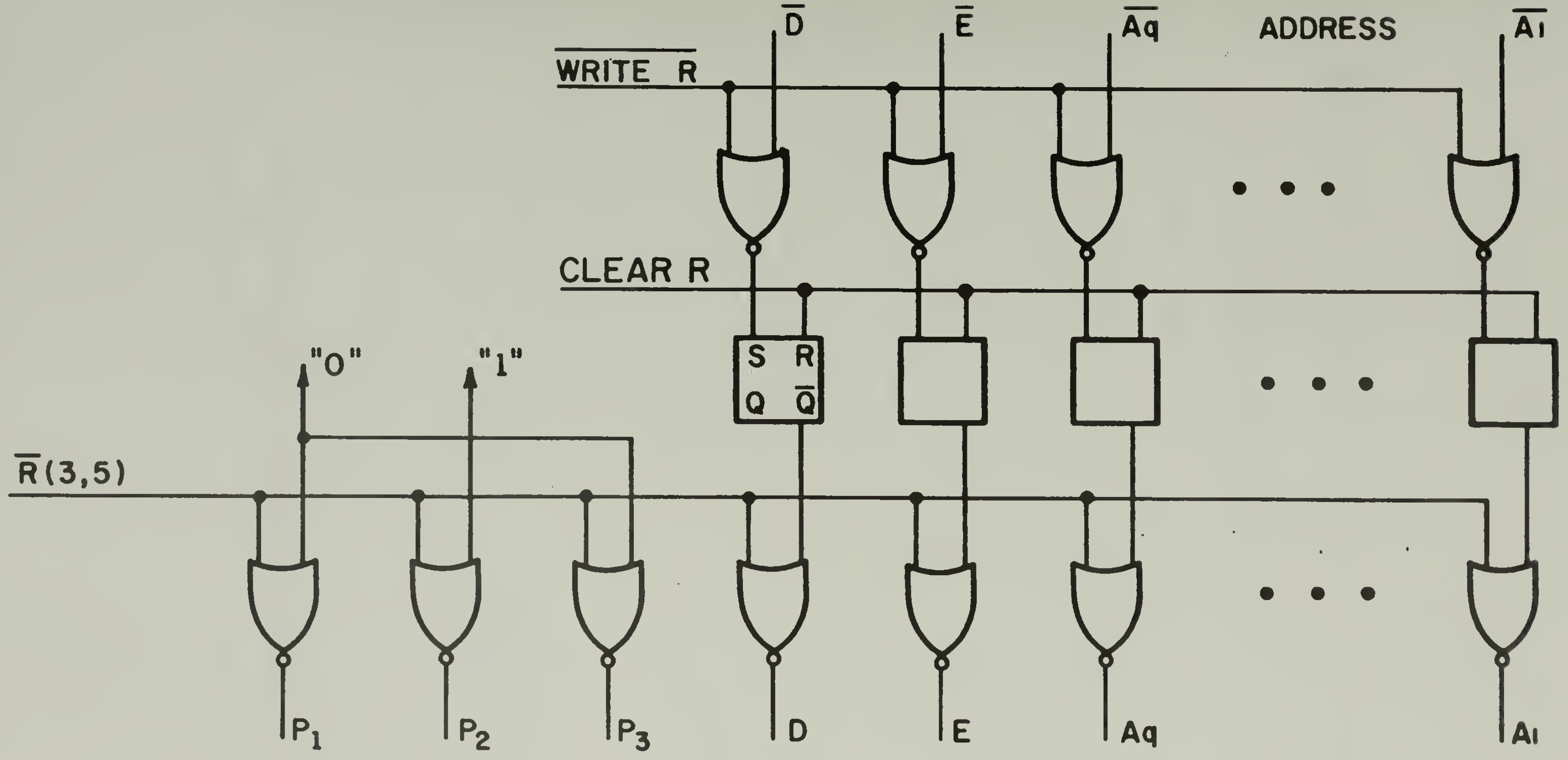
Optional logic can be added on the input side of the register for reducing logical expressions to a single boolean value. Section 2.3.2.2 mentioned the practicality of providing this for log exs consisting of two operands. Assume that each relational expression in a log ex has been converted to a boolean variable according to the rules in section 2.3.2.1. Then a two operand log ex can be written as:

(BV1)LØ1(BV2)



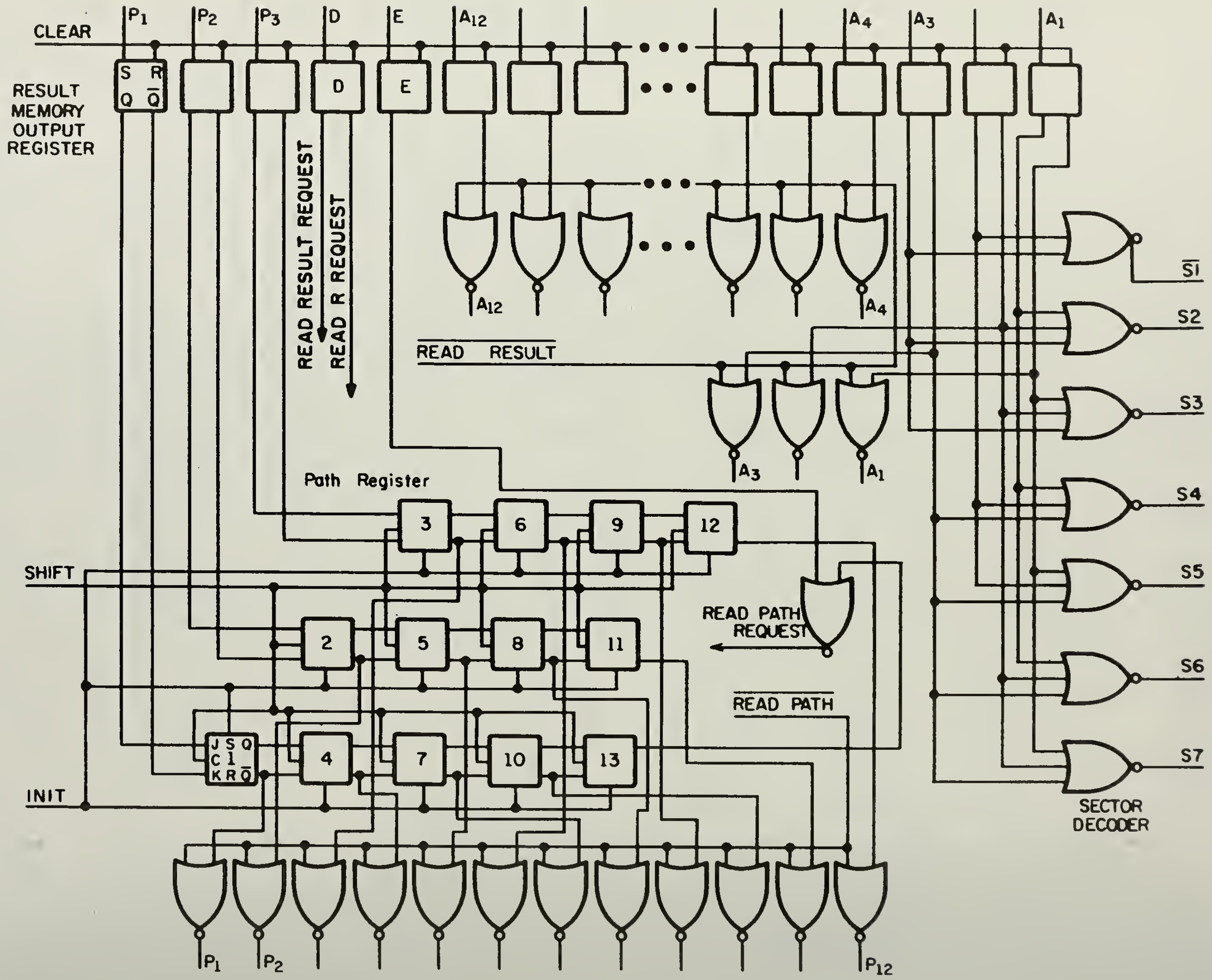
(a) Input Node Register and Tree Decoder

Figure 2.9. Logic Design for a Three Level Decision Tree Processor



(b) Result Memory, Word Five

Figure 2.9 (continued). Logic Design for a Three Level Decision Tree Processor



(c) Result Memory Output Register, Path Register, and Sector Decoder

Figure 2.9 (continued). Logic Design for a Three Level Decision Tree Processor where BVi is a boolean variable in either true or complement form, and LØ1 is the logical operator. Let the operator be either AND or OR. Let LØ1 = 1 when the operator is AND. Let LØ1 = 0 for an OR operator.

The function of the two boolean variables, f2, can be implemented by the equation:

$$\begin{aligned} f2 &= (BV1 \cdot BV2 \cdot L\emptyset 1) \vee (BV1 \vee BV2)\overline{L\emptyset 1} \\ &= (BV1 \cdot BV2) \vee (BV1 \cdot \overline{L\emptyset 1}) \vee (BV2 \cdot \overline{L\emptyset 1}). \end{aligned}$$

A NOR logic design for f2 is given in Figure 2.10. Output $\overline{f2}$ is needed for gating by $\overline{\text{LOAD N}}$ to give the signal, f2e, to the input node register.

Input logic for larger expressions can be designed but the usefulness is questionable due to infrequent use of such expressions by programmers. One such simple design is the cascaded application of the two operand module. This design is shown in Figure 2.11 for a three operand expression, f3. This design does imply a grouping of the first two variables before applying the second operator. That is,

$$f3 = [(BV1)L\emptyset 1(BV2)]L\emptyset 2(BV3).$$

2.4.2 Tree Decoder

This decoder identifies the path through the tree by considering node register data and sector selection control. The result is used to read a word from the result memory. Figure 2.9 (a) has the design for a three level tree decoder.

Equation 2.1 is the logic implemented for a single node of the decoder. The equation is applied recursively to build a decoder with the desired number of levels (value of ℓ). Equation 2.1 (a) is used for the false branch out of the node and 2.1 (b) is used for the true branch.

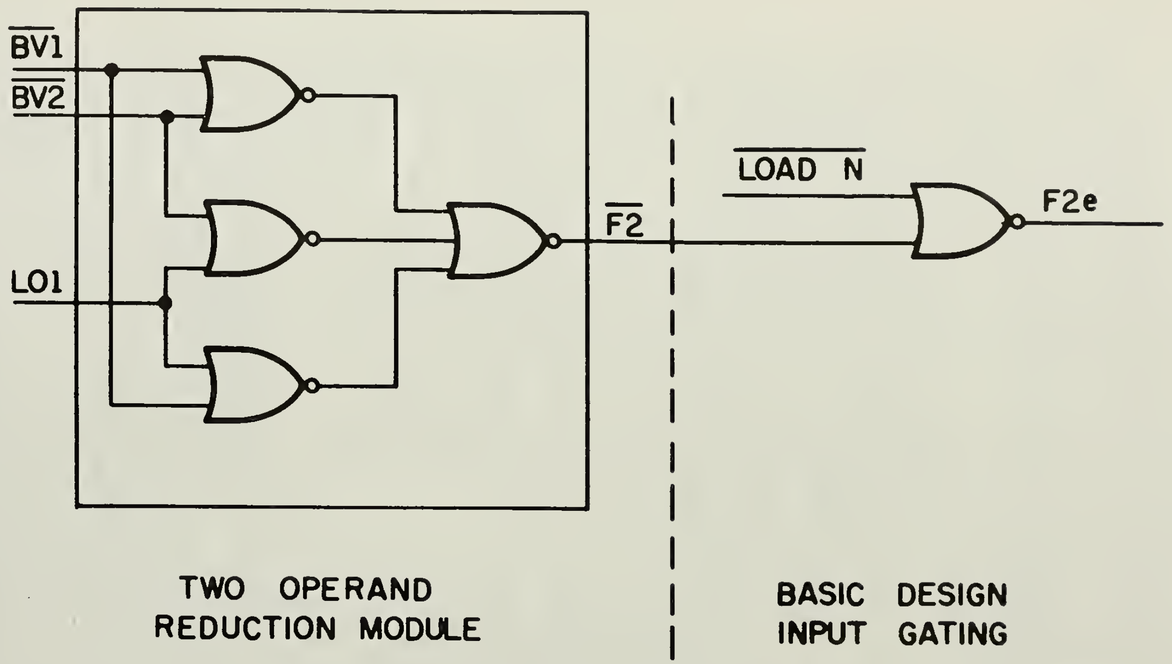


Figure 2.10. Logical Expression Reduction

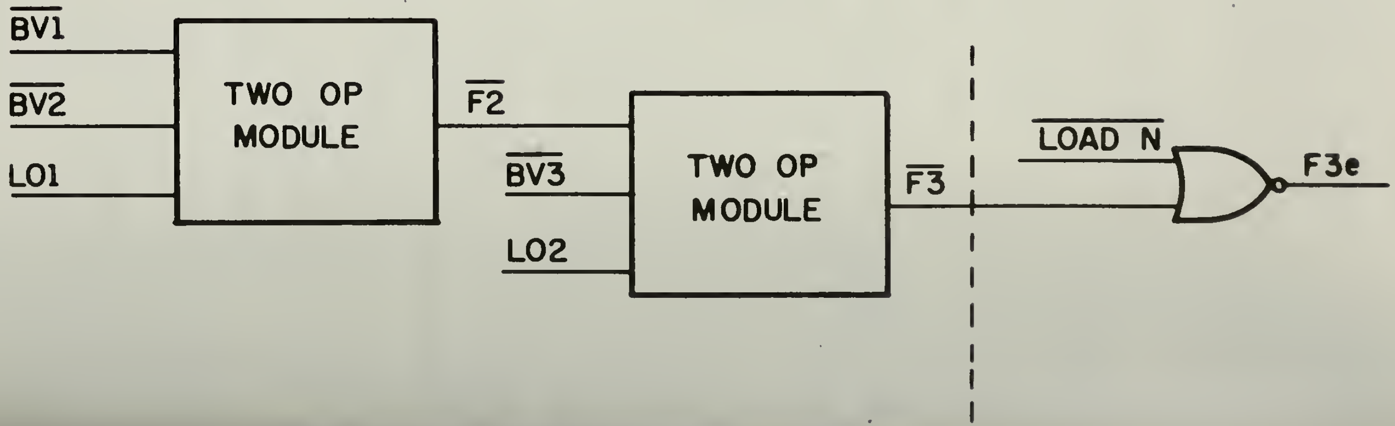


Figure 2.11. Cascading Reduction Modules

$$R(\ell,2i) = \bar{n}(2^{\ell-1}+i)[S(2^{\ell-1}+i) \vee R(\ell-1,i)] \qquad \text{Equation 2.1 (a)}$$

$$R(\ell,2i+1) = n(2^{\ell-1}+i)[S(2^{\ell-1}+i) \vee R(\ell-1,i)] \qquad \text{Equation 2.1 (b)}$$

$$1 \leq \ell \leq k$$

$$0 \leq i \leq (2^{\ell-1}-1)$$

$$R(0,i) = 0$$

Examples of Equation 2.1:

$$R(1,0) = \bar{n}(1)\cdot S(1)$$

$$R(1,1) = n(1)\cdot S(1)$$

$$R(2,0) = \bar{n}(2)[S(2) \vee R(1,0)]$$

$$= \bar{n}(2)[S(2) \vee \bar{n}(1)\cdot S(1)]$$

$$R(2,1) = n(2)[S(2) \vee R(1,0)]$$

$$R(2,2) = \bar{n}(3)[S(3) \vee R(1,1)]$$

$$= \bar{n}(3)[S(3) \vee n(1)\cdot S(1)]$$

$$R(2,3) = n(3)[S(3) \vee R(1,1)]$$

Figure 2.12, picturing a labeled two level tree, will clarify the equation. It is clear, for example, that for R(2,3) to be true n(3) must be true. One of two other conditions must be satisfied. Either S(3) must be the selected sector or n(1) must be true with S(1) selected.

2.4.3 Result Memory

When the tree decoder has selected a result the decision processor must convert that result into the path taken through the tree and the address of the next program statement. This is accomplished by reading the result memory word corresponding to the decoder result.

Since only one path leads to a given result the path data can be hard wired in each memory word. A bit per level is required.

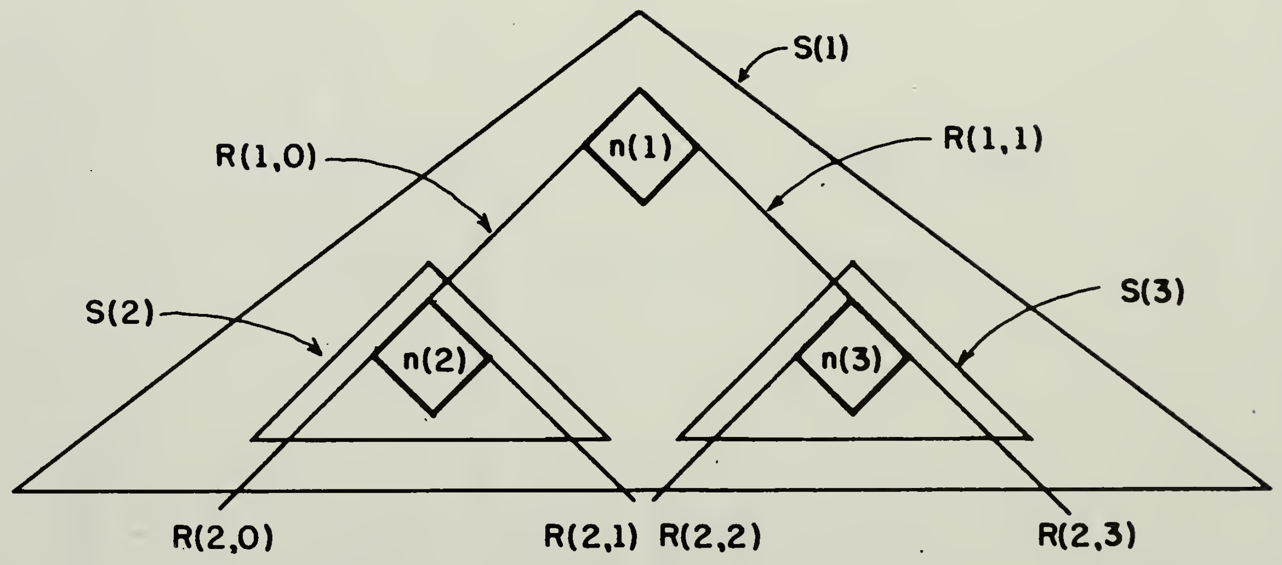


Figure 2.12. Tree with Decoder Equation Labels

Three possibilities exist for the next program statement. The next statement could be a node currently in the input node register in which case the address is interpreted as a sector address selection for another cycle of decoding. For the second possibility the next statement could be a node in the same decision tree which is not currently in the node register. In this case the node register is reloaded. Evaluation of trees which cannot totally be mapped into the processor requires this iteration. The third possibility is that an exit has been reached; that the tree has been evaluated.

Distinction between the three possibilities is made by the D bit which is zero when the address is to be interpreted as a sector for another cycle, and the E bit which is one when an exit has been reached. For all three cases the address portion of the memory word and bits D and E are loaded from the arithmetic processors for each iteration. A memory word is detailed in Figure 2.9 (b).

Memory output is stored in the result memory output register. This register can be separated into path, control, and address fields. The path field is the input to the path register. The control field accepts bits D and E. The address field is a processor output register when bit D is one; it is a sector address when bit D is zero. The example register is in Figure 2.9 (c).

This memory is a large part of the total tree processor in terms of gate and flip flop counts so an option is mentioned here to reduce the counts. For maximum flexibility and simplicity of operation the design provides for a full tree. This is sensible for most of the processor since the hardware per node is reasonable. However the hardware may be unreasonable and unnecessary for the memory. The memory requirement is for one word per exit. The question is how many exits will the vast majority of trees, processed with a k level processor, have. For $k = 8$ the maximum number of exits is 256 while the majority of trees may have less than 64, or even 16, exits. Thus an indirect addressing scheme may be used in which a $j < k$ bit address is selected by the tree decoder and it points to the result in a 2^j word memory.

2.4.4 Path Register

This shift register accumulates data that identifies the path through the tree. Each cycle of the decoder inserts k bits in the least significant end of the register after shifting the previous contents k bits toward the most significant end.

In the initial state the least significant bit, lsb, is set while all other bits are reset. The register is full when the most significant bit becomes set by virtue of the initial lsb propagating to it. At this time the path data must be transferred to the processors and the register initialized. The register is also read and initialized when an exit is reached.

The length of the register is somewhat arbitrary. At the minimum it must have k bits. Read and initialize operations would take place at each cycle eliminating the need for shift capability and the initial set of the lsb. A longer register allows more cycles before it is necessary to read and initialize. The register should have $c \cdot k+1$ bits where c is the desired number of cycles. The extra bit receives the propagated lsb to signal a full register.

Logic for this register is shown in Figure 2.9 (c) for a three level tree with c = 4.

2.4.5 Sector Decoder

Straightforward decoding of the result memory output register address field to a signal corresponding to an individual sector address is used. The logic is enabled at all times influencing decoding of input node register data. The initial state of the sector address field decodes as sector one, such that the entire processor tree is used for the first cycle of each iteration. The three level sector decoder is shown in Figure 2.9 (c).

Table 2.2 in section 2.4.6 shows that the gate count for sector decoding increases rapidly as the number of levels increases. This is due to the exponential increase in the number of sectors and the need for additional gates to handle fan-in of the address bits for decoding each sector. Section 2.5.4 discusses the need for sector control at levels near the bottom of the tree and concludes that the performance return on gates invested decreases as control is extended to the lower levels. Thus a design option is to provide sector decoding for a number of levels called _sector depth_, sd, less than k.

2.4.6 Gate Counts and Timing

Practicality of the processor is determined by cost and performance. One indication of cost is the count of gates used. This is not the most significant factor in current technology but it is relevant and readily gathered. Performance will be measured by gate delays for an iteration and a cycle of the processor.

Table 2.2 gives the gate count for each component of the processor for several values of k. A distinction is made between gates and flip-flops.

Levels	Node Register	Tree Decoder	Result Memory (12 bit address)	Result Mem. Output Reg.	Path Reg. (c=4)	Sector Decoder	TOTALS Including Result Mem.	Excluding Result Mem.
4	15 gates	44 g	512 g	12 g	18 g	15 g	616 g	104 g
	15 f/f		224 f	18 f	17 f		272 f	48 f
6	63 g	188 g	2176 g	12 g	26 g	62 g (sd=5)	2591 g	415 g (sd=6)
	63 f		896 f	20 f	25 f	126 g (sd=6)	1004 f	108 f
8	255 g	764 g	9816 g	12 g	34 g	126 g (sd=6)	11646 g	1830 g (sd=8)
	255 f		3584 f	22 f	33 f	254 g (sd=7)	3894 f	310 f
						765 g (sd=8)		
k	2^k-1 g	$3(2^k-1)-1$ g	$2^k(28+k)$ g	12 g	4k+2 g	$\left\lceil\frac{sd-1}{3}\right\rceil(2^{sd}-1)$ g		
	2^k-1 f		$2^k(14)$ f	14+k f	4k+1 f			
Increment due to adding level k+1	2^k g	$3(2^k)$ g	$2^k(30+k)$ g	-	4 g	Dependent on		
	2^k f		$2^k(14)$ f	1 f	4 f	sector depth		

Table 2.2. Decision Processor Logic Counts

							TOTALS	
Levels	Node Register (1)	Tree Decoder (2)	Result Memory (3)	Result Mem. Output Reg. (4)	Path Reg. (5)	Sector Decoder (6)	Cycle 6+2+3+4	Iteration 1+2+3+4
4	1 gate	7 g	1 g	1 g	1 g	1 g	10 g	10 g
	1 f/f			1 f	1 f		1 f	2 f
6	1 g	11 g	1 g	1 g	1 g	2 g	15 g	14 g
	1 f			1 f	1 f		1 f	2 f
8	1 g	15 g	1 g	1 g	1 g	2 g	19 g	18 g
	1 f			1 f	1 f		1 f	2 f
k	1 g	2k-1 g	1 g	1 g	1 g	1 g: $k \leq 4$	2k+3 g	2k+2 g
	1 f			1 f	1 f	2 g: $5 \leq k \leq 16$	1 f	1 f
Increment due to adding level k+1	-	2 g	-	-	-	- : $k \neq 4$	2 g	2 g

Table 2.3. Decision Processor Logic Delays

General formulas are given for a k level processor and for the incremental count due to adding level k+1.

Path register design was fixed at c = 4 (storage for four cycles) for the table. Address field length for the result memory and its output register was fixed at 12 bits.

Since the result memory makes up such a significant part of the count, two columns are used for totals. The first gives the count including the result memory. The second is the total excluding the memory. An indirect addressing scheme to reduce the memory requirements was mentioned in section 2.4.3 but gate counts for that option have not been tallied.

Table 2.3 gives the delay in gates and flip flops at each component. The total delay is not the sum of the component delays since some component actions occur simultaneously.

Timing for the first cycle of an iteration of the processor includes the node register, tree decoder, result memory output register, and the final output. If the path is to be read, the path register delay takes the place of the output delay.

Timing for a cycle other than the first cycle in an iteration begins with sector decoding, does not require a reload of the node register, then proceeds as does an iteration. That is, sector decoding time replaces node register loading time.

2.5 Processor Operation and Performance

2.5.1 An Example of IF Tree Processing

This section follows an IF tree from the program to its execution. The program segment is written in a PL/I like language and uses b and f to represent boolean and arithmetic functions.

The program segment IF tree:

```
IF b(A) THEN
  IF b(B) THEN
    DO; C = f1;
        IF b(C) THEN
          DO; K = f2;
              IF b(D) THEN R;
              ELSE S;
          END;
        ELSE  IF b(E) THEN T;
              ELSE IF b(H) THEN
                         DO; H = f3;
                             IF b(H) THEN U;
                             ELSE V;
                         END;
                    ELSE W;
    END;
  ELSE DO;  C = f4;
            IF b(C) THEN X;
            ELSE Y;
       END;
ELSE Z;
```

Figure 2.13 is a pictorial representation of the program. Nodes and branches are labeled with their binary names.

Apply Algorithm 2.2 to reduce the IF tree to a block of assignment

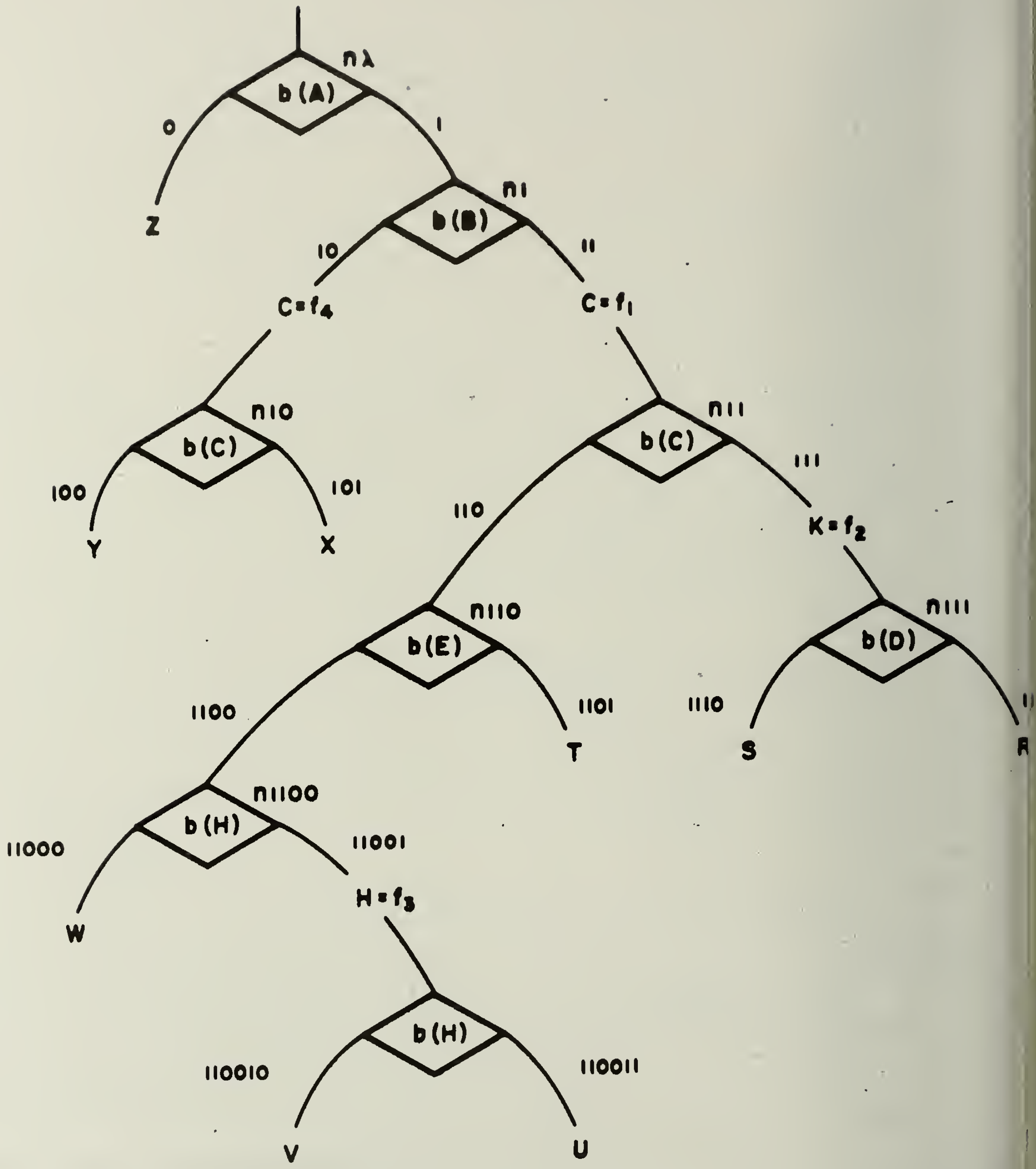


Figure 2.13. IF Tree Corresponding to Example Program Segment

statements followed by a decision tree. Algorithm 2.1 converts the logical arguments to boolean variables symbolized with lower case letters as in "a = b(A)". Figure 2.14 shows the block of assign sts and the six level decision tree that results. Descriptors have been attached to variable names where needed.

For this example assume the processor is capable of evaluating four level full trees. Mapping the first four decision tree levels into the processor gives four free nodes including a two level sector. Fold the leftover two levels into this sector, as in Figure 2.15, so the tree can be evaluated in one iteration of either one or two cycles.

The original decision tree had six levels and thus required only six bits to specify a bottom level result. Now each cycle of the processor produces a four bit result. If the first cycle result is 1100 indicating S5 as the next statement, a second cycle is needed. The final result has eight bits. For consistency the binary address of the sector which receives the fold must be inserted in all descriptors which are part of the fold. The extra bits are inserted following the bits that identify the exit which points to the fold. In this example exit WVU is 1100 and the sector which receives the fold is 01. The descriptor for F changes from 11001 to 1100011. Paths which exit at W, V, and U likewise have 01 inserted in their binary label. All of the preceding work is performed at compile time.

In execution the block of assign sts can be done in parallel including those assignments that result in boolean node values. All node values are loaded into the node register. Addresses of next program locations and bits D and E are loaded in the result memory as shown in Figure 2.16.

Cycling the processor once will yield an exit unless path 1100 is

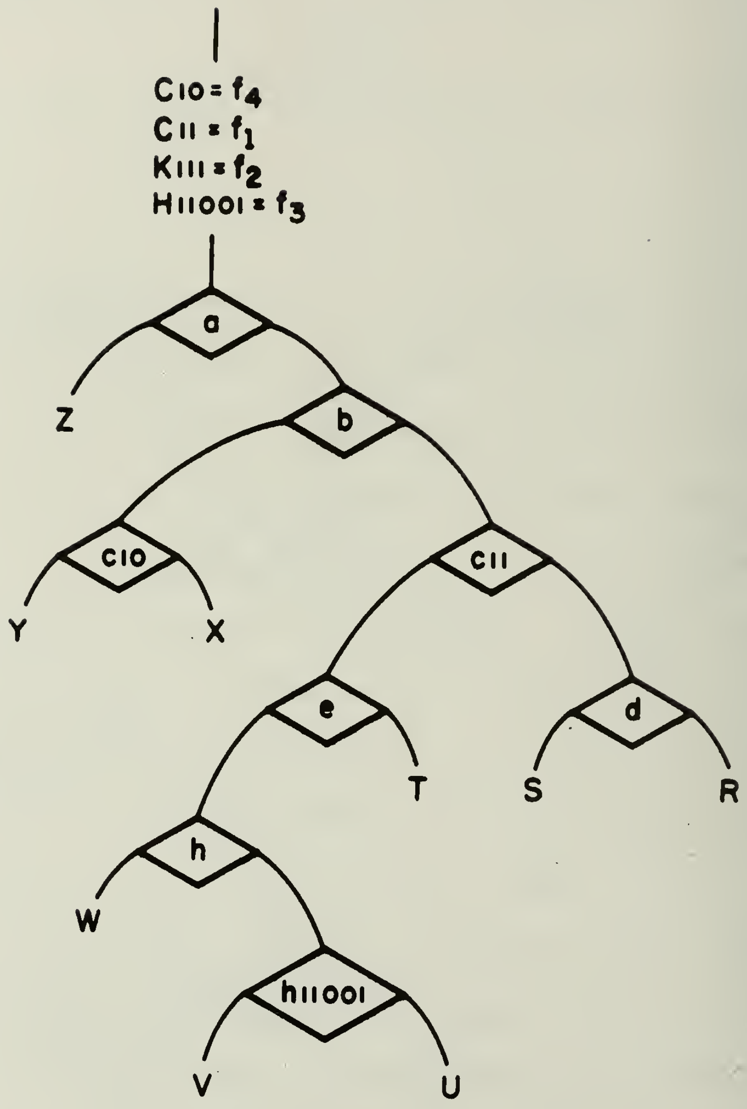


Figure 2.14. Decision Tree Derived from Figure 2.13

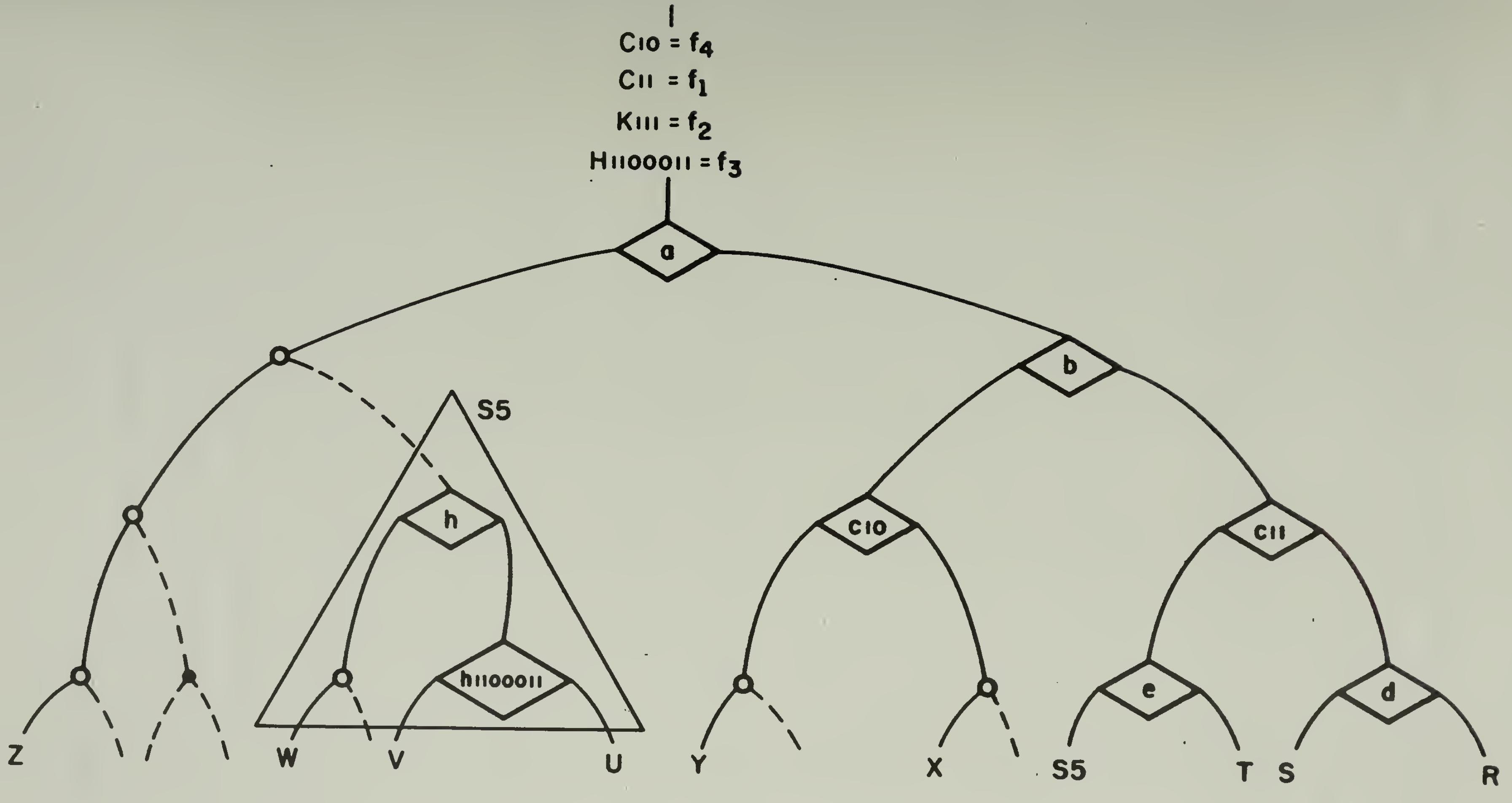


Figure 2.15. Example Decision Tree Mapped into Processor Structure

taken. In this case the address field is decoded as a sector for control of the second cycle. The result of the second cycle will point to a memory location with exit W, V, or U.

Path	D	E	Address
0000		1	Z
0001			
0010			
0011			
0100		1	W
0101			
0110		1	V
0111		1	U
1000		1	Y
1001			
1010		1	X
1011			
1100	1		S5(=01)
1101		1	T
1110		1	S
1111		1	R

Figure 2.16. Result Memory Contents for Example IF Tree

Assume two cycles were necessary and V was the selected exit. After the first cycle the path register was loaded with 1100. The contents were shifted after the second cycle to receive the next four bits, 0110. The total path is 11000110.

At this point the next program location has been determined. What remains is selection of variables for permanent assignment from the block of assign sts removed from the IF tree. Statements selected are those on the path taken. Identification is by descriptors which match the path bit pattern from the most significant end. Thus C = C11 and F = F1100011 are selected. The example is concluded.

2.5.2 Mapping: Folding and Multiple Input Nodes

Control over processor tree sectors is provided to extend the usefulness of the processor. Decision trees with k of fewer levels and single input nodes map into the processor in a straightforward way. Other trees may need folding, special mappings, and sector control to achieve satisfactory performance.

2.5.2.1 Folding

Examples of folding have been given in previous sections, especially 2.5.1. In this section some observations and limits are noted. The assumption is made here that the decision tree to be folded has more nodes than the processor. That is, the decision tree is not exhausted by folding.

Folding can be applied when free nodes exist. A free node occurs as a consequence of a decision tree node at a level $i \leq (k-2)$ with an exiting branch. First a transmit node is formed at level i+1. The successors, at i+2, of the transmit node are a transmit node and the free node. A picture of this situation is given in Figure 2.4 (b).

Several facts concerning free nodes can be stated.

Fact 1. All nodes in the sector of successors from a free node are free, prior to any folding operation in the sector.

Fact 2. Level three is the highest level at which a free node can occur.

This can be observed in Figure 2.4 (b).

Fact 3. A branch which exits at level i requires (k-i) transmit nodes and introduces a sector of free nodes at each level from (i+2) to k.

Fact 4. A free sector at level i can accept a tree of length (k-i+1).

Fact 5. The immediate successors of a transmit node are one transmit node and one free node.

An algorithm for folding IF tree nodes into the decision processor is sketched here. The mapping goal is to convert free nodes to decision nodes while minimizing the number of transmit nodes. The execution goal is to minimize the number of cycles required for execution. Assuming no knowledge of which exits are most likely the two goals are reasonably met in the following way. First determine the number of levels in the largest sector of free nodes. Then, from the highest decision tree level with unmapped nodes, select the node which is the root node of the largest unmapped subtree. Map that node and its successors into the free sector for the appropriate number of levels. Continue the above steps until all free nodes are used or the decision tree is exhausted.

A tree of a particular shape is now defined. Let a <u>linear decision tree</u> be one in which there is only one node per level.

The largest number of nodes that can be processed in one iteration is the number of nodes in the processor tree. A full, k level decision tree is required for this situation. The largest number of levels that can be processed in one iteration occurs with a linear decision tree. This statement is due to the following fact.

Fact 6. Every tree includes a linear tree as a subset of the nodes in the tree. The subset consists of one node from each level.

Now determine the largest number of levels per iteration. A linear decision tree has an exit at every level. After mapping the first k levels examine the free nodes. According to Fact 3 there are sectors of free nodes at levels 3,4,...,k due to the level one exit; at levels 4,5,...,k due to the level two exit; etc. Remaining segments of the tree are mapped into free sectors.

Using Fact 4, a tree segment of length (k-2) can be mapped into the sector at level three. This produces additional free sectors at levels 5,6,...,k; 6,7,...,k; etc.

Continuing this enumeration leads to equation 2.2 for the maximum number of levels which can be mapped into a k level processor. Each term is the number of levels per fold mapped into a free sector. The count of terms is then the maximum number of cycles required to evaluate the tree.

$$L_k = k+(k-2)+(k-3)+(k-3)+(k-4)+(k-4)+(k-4)+(k-4)+\dots+1 \qquad \text{Equation 2.2}$$

Equation 2.2 is rewritten below with the terms grouped.

$$L_k = k+(k-2)+2(k-3)+4(k-4)+8(k-5)+\dots+2^{k-3}(1)$$

$$= k+\sum_{i=3}^{k} 2^{i-3}(k-i+1)$$

For a six level processor

$$L_6 = 6+4+2(3)+4(2)+8(1)$$

$$= 32.$$

That is, a 32 level linear decision tree can be evaluated in one iteration on a six level processor.

Two bounds are now known for single iteration processing of trees on a k level processor. The maximum number of nodes is 2^k-1. The maximum number of levels is given by Equation 2.2. Section 2.5.3 will establish a third bound: the maximum number of nodes for which processing in a single iteration can be guaranteed.

2.5.2.2 Multiple Input Nodes

All previous discussion has been concerned with trees in which every node had only one input. In this section some of the ways nodes can have multiple inputs are mentioned along with suggested means for dealing with them. The suggestions are all compile time operations and have not been examined thoroughly.

For a single node within an IF tree to have multiple inputs means there is a way to reach the node other than by the branch from the node above. Consider first an input from outside the IF tree. Unless the multiple input node maps onto the root node of the processor, the program must be modified to let the processor perform properly. Recall that the sector controls initially select sector one, the whole processor tree. The compiler must compensate for this by inserting dummy IF statements in the program to build a path from the root node to the node in question.

For example, assume an IF statement labeled HERE maps onto processor node five and the program includes a GO TO HERE statement. Let the compiler insert an IF 0 THEN IF 1 preceding the GO TO to build a path to node five.

As a second instance of a multiple input node consider branch b which does not go to node b or exit b but becomes a multiple input for another node. That is, a transfer is made within the IF tree and the IF tree becomes a network rather than a tree. Connections of this type do not exist in the processor. This situation is complicated by the necessity for maintaining information on the path taken.

A solution is to duplicate the sub-tree descending from the multiple input node, where needed, to produce a network free tree. Variable descriptors, as mentioned in relation to assignment statement movement, must be attached according to the shape of the tree with duplicated sub-trees.

A loop exists if a path returns to a predecessor node. The loop can be valid (non-infinite) if it includes an assignment statement which can change the decision made at some node in the loop. Duplicating sub-trees is useful only to the extent of the single iteration capability of the processor. Loops can be expanded to fill the processor. Further expansion should be dependent on the results of the iteration.

2.5.3 Processor Efficiency

Examination of many programs has shown that decision trees tend to be sparse rather than full. This would seem to indicate many transmit nodes in the processor and inefficient use of the hardware. It can be shown however that a processor with $n = 2^k - 1$ nodes and complete sector control can process an $(\frac{n+1}{2})$ node decision tree of any shape in one iteration. That is, regardless of tree shape, more than half of the processor nodes are available for use as decision nodes. For an ℓ level decision tree with $\ell > k$, folding is obviously required.

Define processor efficiency as the percentage ratio of decision plus free nodes to the total number of nodes in the processor. Free nodes are available for decision use and thus are grouped with decision nodes.

Statement: Processor efficiency for a k level processor can always be greater than 50% for every iteration required to evaluate a decision tree which has at least k levels.

When the decision tree has fewer than k levels, processor nodes are used to transmit results to level k rather than make decisions. In this situation the processor capacity is not fully used.

Proof of the Statement:

The root is by definition a decision node. The three possibilities for successor nodes of a decision node will be examined.

Case 1. Both immediate successors are decision nodes.

This clearly represents 100% efficiency locally.

Case 2. One immediate successor is a transmit node, the other a decision node.

Using Fact 4 from section 2.5.2.1 construct Figure 2.17. This

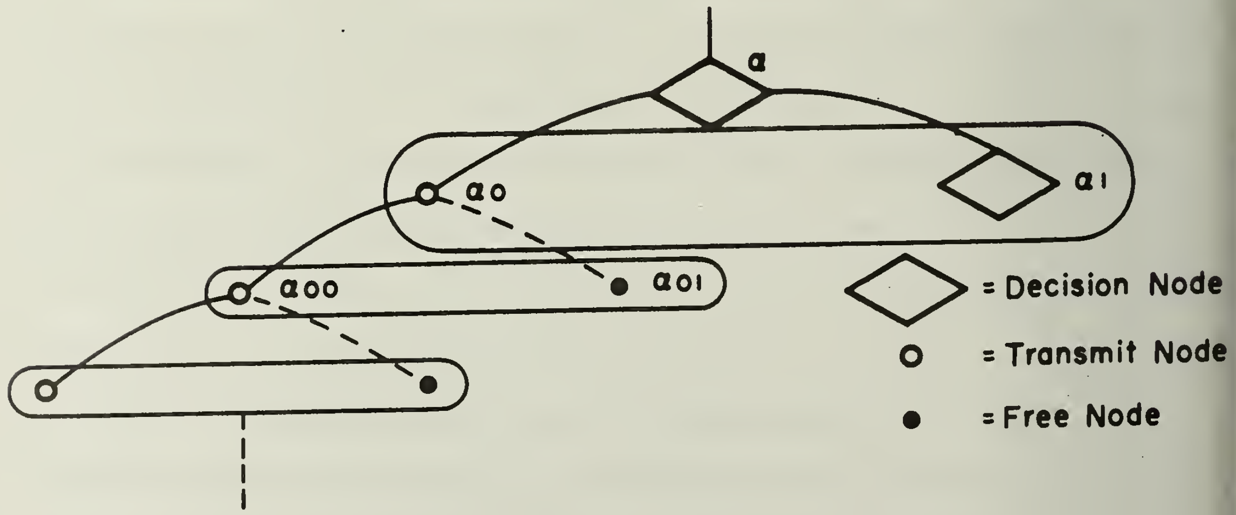


Figure 2.17. Transmit Node Pairings

figure represents the Case 2 successors of any decision node, α. A pairing can be made such that each transmit node has a corresponding decision or free node without involving α in the pairing. If α is the root node it is always unpaired. If α is not the root it may be paired with the other successor of its predecessor as $\alpha 1$ is paired with $\alpha 0$.

Case 3. Both immediate successors are transmit nodes.

The three nodes can be remapped equivalently as shown in Figure 2.18. Remapping yields a free node which can be paired with the transmit. If the decision node was previously paired with a transmit in Case 2, let that pairing continue.

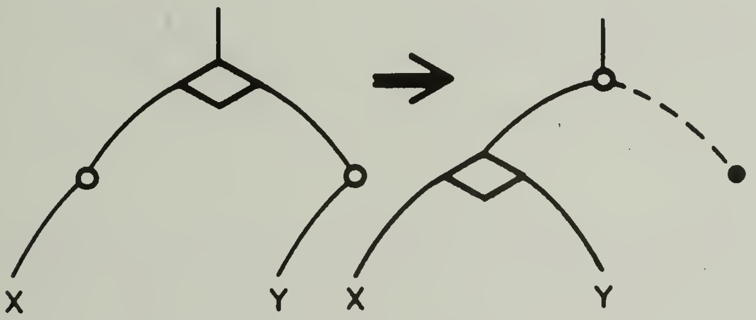


Figure 2.18. Remapping for Better Efficiency

Applying the three possibilities to every decision node results in a pairing for every transmit node with at least the root decision node left unpaired. The efficiency is therefore greater than 50% by at least one decision node for which there is no paired transmit node. This concludes the proof.

The third bound mentioned in section 2.5.2.1 is established by the above statement. For a k level processor there are $n = 2^k - 1$ nodes. At least $\frac{n+1}{2} = 2^{k-1}$ can always be decision nodes. Thus 2^{k-1} is the maximum number of decision nodes per iteration which can be guaranteed to map into the processor.

A note on the significance of this is useful. The maximum number of decision nodes which can be guaranteed to map into the processor is essentially a minimum number of decision nodes per iteration, excluding trees which do not use the capacity of the processor. A linear decision tree is the only one for which the minimum holds. The linear decision tree is also the one for which the maximum number of levels can be processed. Now let $k = 6$. The minimum number of nodes is 32. The maximum number of levels is 32, from Equation 2.2 in section 2.5.2.1. Thus it would take the uncommon program segment consisting of 32 sequential IFs for the minimum decision node bound to apply.

2.5.4 <u>Performance Tradeoffs Between Iterations and Cycles</u>

Gate and flip flop delays for the first cycle of an iteration and for each succeeding cycle are nearly the same, from Table 2.3. In operation however, the first cycle of an iteration requires communication with arithmetic processors whereas succeeding cycles are internal to the decision processor. The real times are thus not nearly equal and for this section the assumption is made that an execution time for the first cycle of an iteration takes M times as long as for any succeeding cycle.

Cycles are the result of folding trees into free sectors. At the lower levels of the processor the nodes per sector are few, which means that the nodes evaluated per cycle are few. At some point it becomes more time effective to discontinue folding in small tree segments and resort to a new iteration. That point is apparently where the probability of reaching an exit is less in the next M cycles within the current iteration, than in the first cycle of the next iteration. If all exits are equally likely it is a simple matter to count their occurrence if mapped as the next M cycles versus one cycle in the next iteration.

Linear trees are examined to demonstrate the tradeoff. Equation 2.2 is a summation in which each term is the number of nodes mapped per fold. Table 2.4, which follows, uses that equation to determine the entries in the nodes per fold column.

levels k	processor nodes 2^k-1	linear tree nodes 2^{k-1}	single iteration nodes per fold (terms of Equation 2.2)
2	3	2	2
3	7	4	3,1
4	15	8	4,2,1,1
5	31	16	5,3,2,2,1,1,1,1
6	63	32	6,4,3,3,2,2,2,2,1,...,1
7	127	64	7,5,4,4,3,3,3,3,2,...,2,1,...,1
8	255	128	$8,6,5,5,4,4,4,4,\underbrace{3,\ldots,3}_{8},\underbrace{2,\ldots,2}_{16},\underbrace{1,\ldots,1}_{32}$

Table 2.4. Nodes Evaluated on Succeeding Cycles for Linear Decision Trees

If more than one iteration is used, the larger numbers at the beginning of the nodes per fold series are reused, eliminating the long strings of ones and twos. Table 2.5 is a compilation of the cycles required to evaluate various length linear trees using multiple iterations. The entries are derived from Table 2.4 as in the following example. The entry at k = 6 for three iterations is determined by first noting that the linear tree to be evaluated has 32 nodes. If at most three iterations are to be used, $\left\lceil \frac{32}{3} \right\rceil$ = 11 nodes must be examined in at least one iteration, say the first. From Table 2.4, in one iteration the first cycle examines six nodes, the second four, and the third three bringing the total to 13. Three cycles were required to get the total above 11 and three becomes the first number in the table entry being determined. There are 32-13 = 19 nodes for the remaining two iterations. One of them must examine at least 10 nodes and the other the remainder. Again from Table 2.4 it can be seen that two cycles cover 10 nodes, so two becomes the second number in the entry and nine nodes remain. Two cycles pick up the remaining nodes so the last number in the entry is also two.

Behind these tables is the goal of selecting the best compromise between iterations and cycles. The best operating point is a function of the multiplier, M, relating the execution times of iterations and cycles.

The time required to evaluate a decision tree on the processor is the time for the first cycle of each iteration plus the time for all succeeding cycles. The first cycle includes communication with the arithmetic processors and has been defined as taking M times longer than succeeding cycles to execute.

Cycles per Iteration	Iterations Used to Evaluate a 2^{k-1} Level Linear Decision Tree						
Processor Levels (k)	1	2	3	4	6	10	16
2	1	1	1	1	1	1	1
3	2	1,1	1,1	1,1	1,1	1,1	1,1
4	4	1,1	1,1	1,1	1,1	1,1	1,1
5	8	2,2	2,1,1	1,1,1,1	1,1,1,1	1,1,1,1	1,1,1,1
6	16	4,4	3,2,2	2,2,1,1	1,1,1,1,1,1	1,1,1,1,1,1	1,1,1,1,1,1
7	32	8,8	5,5,4	3,3,3,3	2,2,2,2,2,1	1,1,1,1,1,1,1,1,1,1	1,1,1,1,1,1,1,1,1,1
8	64	16,16	9,9,9	6,6,6,6	4,4,4,4,3,2	2,2,2,2,2,2,2,2,2,1	$\underbrace{1,1,\ldots,1}_{16}$

Table 2.5. Cycles Required, per Iteration, to Evaluate a Linear Decision Tree

Let I represent the number of iterations used, corresponding to columns of Table 2.5. Let C_I represent the total number of cycles used for a particular value of I. Then I is the number of first cycles and C_I-I is the number of succeeding cycles required to evaluate a decision tree. The time required is thus proportional to $M \cdot I + C_I - I = (M-1)I + C_I$. This function is graphed in Figure 2.19 for k = 8 and M = 1,2,3, and 4. Execution time is in units equivalent to the time required to cycle the decision processor. The operating point to select is the one which minimizes the execution time. Thus for M = 1 it is best to use 16 iterations of one cycle each, whereas for M = 3, four iterations of six cycles each is best.

This section has dealt with the fringe case of linear decision trees to simplify data gathering for the tables. The principle is applicable to any shape tree.

Sector control, in the form of sector decoding logic and control gates on the tree decoder, is required for every node that can accept a fold. The return, in decreased execution time, seems non-existent at the lower levels except perhaps in some situation where just a few more nodes would complete the tree. The cost, in logic, is high since there are more gates on the bottom level than in the rest of the tree. In conclusion it is recommended that sector control stop at some level above k.

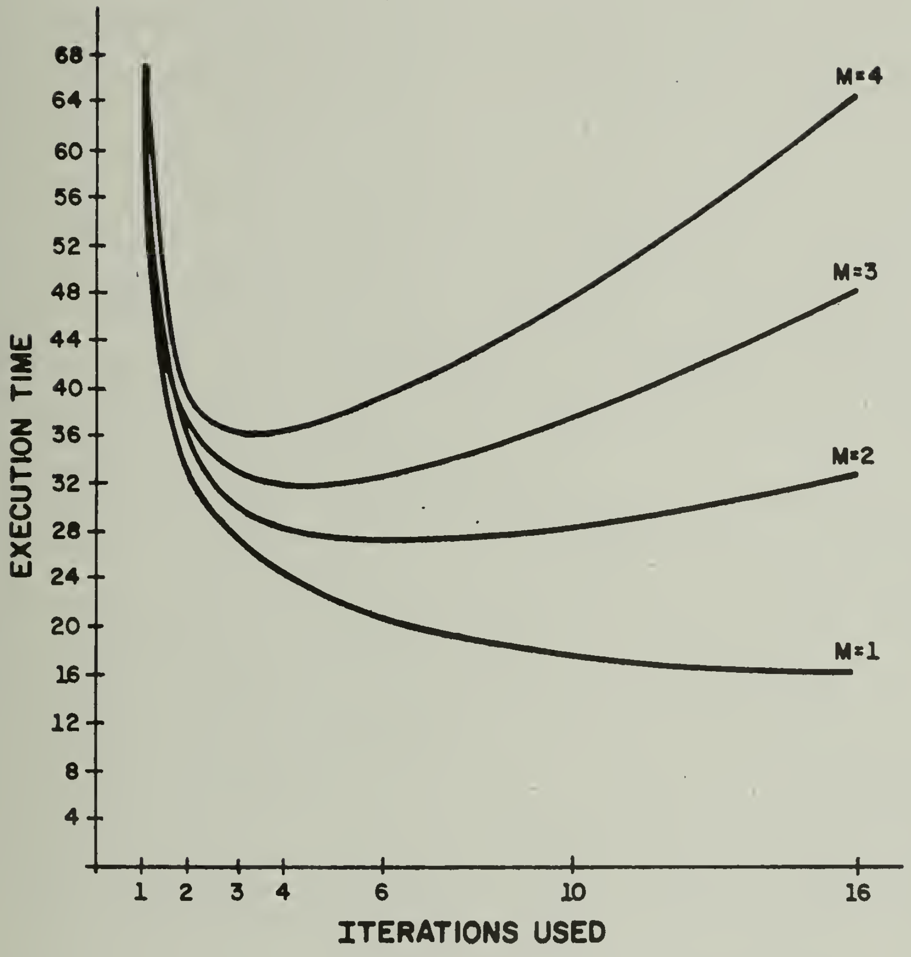


Figure 2.19. Variations in Decision Tree Evaluation Time

3. SIMULATION OF DISCRETE TIME SYSTEMS

This chapter examines some software questions involved in the concurrent processing of discrete time simulation programs. A simulation language is studied to determine the natural parallelism and to develop a machine design philosophy for using that parallelism.

3.1 Languages

Applications programming can be done in any general purpose language. When a particular application comes into frequent use, particular languages tend to be developed to simplify programming. This has been done for discrete time simulation. At the present there are many simulation languages, some being minor variations of others. Among the more widely known are GPSS, the General Purpose Simulation System [7], Simscript [8], and Simula [4].

GPSS is implemented as a fixed set of routines which can be thought of as blocks in a block diagram of the system to be simulated. Simscript, developed at the Rand Corporation, is a powerful language with similarities to Fortran and PL/I but also including features useful in simulating systems that change over time. Simula is an Algol based language that includes Algol as a subset.

These three significantly different languages have characteristics in common that provide for simulation. In all cases a simulated time clock is an inherent device which affects the progress of the simulation. An <u>event</u>, defined as a change in the status of the simulated system, occurs at a scheduled clock time or causes the clock to increment to the event time. Let <u>model</u> mean the representation in a programming language of a system to be simulated. Simulation programs have temporary entities which move through the model.

Temporary entities are those which are not required to exist for the duration of the simulation. The model is described by permanent entities which do exist throughout the simulation. Finally, means to control the progress and interaction of entities and simulated time is provided.

GPSS will be described in more detail since it is used extensively in the remainder of this thesis. There were several reasons for selection of GPSS. A primary reason was availability of the GPSS/360 system, and documentation on its implementation and use. The block diagram structure, essentially making it a higher level language than the procedural languages, clarifies the ideas of the thesis. Also, while GPSS is considerably different from other languages it is neither unique nor unused.

Acceptance of GPSS has prompted the development of several other similar languages. One of these is BOSS, Burroughs Operational Systems Simulator [13]. As stated in that reference "BOSS is a block-diagram-oriented, data base driven simulator program, in the general class of GPSS...". Another is QUICKSIM, an attempt to impart a block diagram structure to Simscript [16]. As a third similar language there is the Computer System Simulator, CSS. An application of CSS, reported in [11], described it as follows: "CSS/360 was used in this study. It is a simulation program.... In concept it is similar to the General Purpose System Simulator (GPSS), differing in one aspect: it is not general, but applies specifically to computer systems."

3.2 A Description of GPSS

The purpose of this section is to present a description of GPSS sufficient for understanding the algorithms and examples that follow. For more complete information refer to the User's Manual [7].

An example of a GPSS program is given in Figure 3.1. The example, intended to show features of GPSS, is not meant to be a meaningful model. A more significant program is listed with comments in the appendix.

A GPSS program is made up of control, definition, and executable statements written in a fixed format, one statement per card. An executable statement is called a block. A fixed set of blocks is provided to represent the components and control of the model to be studied. The format for a block allows up to seven operands. The definition of each block identifies the required and optional operands and their meaning. Each block is in actuality the name of a routine with the operands being parameters. Execution of a block is execution of its routine.

Temporary entities are transactions. Blocks are provided for causing transactions to become active in the model and for removing them from active status. Once activated, a transaction normally moves sequentially through the blocks of the program. The movement of a transaction into a block is a call for execution of the routine that corresponds to the block. Exceptions to sequential movement are caused by blocks which unconditionally or conditionally cause a transaction to transfer to a non-sequential block. These control blocks affect transactions in GPSS much the way GO TO or IF statements affect the instruction counter in procedure oriented languages.

A simulation study typically involves many transactions which interact with each other. It frequently becomes necessary to suspend processing of one transaction and begin work on another. Even so, the movement of one transaction through a GPSS program can be thought of as an execution of the program.

A clock is maintained which automatically updates to the time of the next event. Simulated time has an origin of one and is incremented by integer

```
BLOCK                                                                              CARD
NUMBER  *LOC   OPERATION  A,B,C,D,E,F,G            COMMENTS                      NUMBER
               SIMULATE                                                               1
        *                                                                             2
        *        THIS GPSS PROGRAM IS USED IN EXAMPLES AT SEVERAL POINTS              3
        *        IN THE TEXT.                                                         4
        *                                                                             5
         EXPON FUNCTION   RN2,C12                                                     6
        0     0     .2    .222  .4    .509  .6    .915  .75   1.38  .84   1.83        7
        .9    2.3   .94   2.81  .96   3.2   .98   3.9   .995  5.3   .999  7.0         8
        *                                                                             9
         TERMI FUNCTION   RN8,S5,F                                                   10
        0.2   TERM1 .4    TERM2 .6    TERM3 .8    TERM4 1.0   TERM5                  11
        *                                                                            12
        *        EXECUTABLE GPSS STATEMENTS BEGIN HERE.                              13
        *                                                                            14
1              GENERATE   300,FN$EXPON                                               15
2              ASSIGN     12,FN$TERMI                                                16
3              INDEX      3,10                                                       17
4              SEIZE      P12                                                        18
5              QUEUE      CPU                                                        19
6              SEIZE      CPU                                                        20
7              DEPART     CPU                                                        21
8              ADVANCE    V1                                                         22
             1 VARIABLE   K100*N4                                                    23
        *                                                                            24
        *        THE VARIABLE DEFINITION CARD IS NOT A BLOCK. IT DEFINES AN          25
        *        OPERAND USED BY A BLOCK.                                            26
        *                                                                            27
9              RELEASE    CPU                                                        28
10             RELEASE    P12                                                        29
11             TERMINATE  1                                                          30
               START      10                                                         31
               END                                                                   32
```

Figure 3.1. Example GPSS Program

values. The unit of real time represented by a unit of simulated time is programmer defined. Events are scheduled through the use of the ADVANCE block. Each transaction has an associated set of words including one called "block departure time", BDT. When a transaction moves into an ADVANCE block a time increment is calculated and added to its BDT. Processing of the transaction is suspended until all transactions with smaller BDT values have been processed.

Permanent entities of several types can be used to describe the physical equipment of a model. A facility is an entity which can be occupied by only one transaction at a time. Blocks are provided that let a transaction use (SEIZE, PREEMPT) or relinquish use (RELEASE, RETURN) of a facility. A storage is an entity which can be occupied by more than one transaction. The storage capacity is given for each storage by a definition card. ENTER and LEAVE blocks are provided to let transactions use or relinquish use of a storage. A logic switch is a two state entity used to control the movement of transactions. The LOGIC block results in the setting, resetting, or inverting of a switch.

Other blocks can be used to simulate various queuing schemes, assign values to variables, control transaction movement, group transactions or numbers, gather statistics, and give diagnostic or partial result information.

GPSS provides a set of Standard Numerical Attributes, SNA's, which are names of variables. These variables are selected attributes of GPSS entities. An SNA consists of a one or two letter mnemonic and, in most cases, an index to identify a particular entity. For example, a facility can either be in use or available. The status is given by SNA Fj, where j is the index of the facility. In this example F3 = 0 indicates facility three is available.

The major use of SNA's is in the operand field of blocks. They are in fact the only variables a programmer can use. Symbolic naming of entities is allowed in which case the symbol replaces the index number. Since all variables are SNA's defined in GPSS, all possible variable names and certain of their characteristics are known at compile time.

A characteristic of interest in section 3.3.2 concerns a limitation on accessing certain variables. In particular, each transaction has a priority, PR, and a set of parameters, Pj, $0 \leq j \leq 100$. When PR or Pj are used as block operands they refer only to the transaction which is executing the block. A given transaction cannot use the value of the priority or any parameter of any other transaction as an operand. These SNA's are considered transaction related variables.

System related variables can be accessed by any transaction. An example is the storage location called a savevalue, referenced by Xj. Any transaction executing a block with operand Xj refers to the same physical storage location.

A similar distinction can be made for block types based upon the block definition. The PRIORITY and ASSIGN blocks are used to write values into PR and Pj of the transaction executing the block. They affect directly only the transaction being moved. Other blocks in the same category are ADVANCE, TEST, TRANSFER, etc.

Blocks can be identified that give system variables new values. The SAVEVALUE block writes a value into savevalue location Xj which can be accessed by any transaction. Similarly, ENTER, LOGIC, QUEUE, and other blocks can change the value of system variables.

GPSS provides features for using tapes to store intermediate results of large simulations. The features are not considered in this thesis.

Processing a GPSS program includes assembly, input, execution, and output phases. The execution phase, in which transactions are moving through blocks, is of most importance in this thesis. Execution is described in the next section, 3.3, and the description of GPSS is continued, especially in section 3.3.1.

3.3 GPSS Execution

A major goal of this thesis is to design a multiprocessor system for concurrent execution of individual programs written in a language like GPSS. To better understand the problem and the proposed system, serial execution of GPSS is described. Parallelism within GPSS, and thus the potential for concurrent execution, is studied. The constraints limiting concurrency, as imposed by the simulated time aspect of simulation, are demonstrated.

When simulation is being discussed in this section, simulated time will be referred to simply as "time" whereas real time will be identified as such.

3.3.1 Serial Execution: GPSS/360

Processing in the execution phase consists entirely of moving transactions through blocks. One function of GPSS that simplifies the programmer's effort is the selection of which transaction should be moved and into which block it should be moved. With a single processor only one of the potentially many transactions can be selected. Once a selection has been made, one word of the set of words associated with each transaction identifies the next block to be executed.

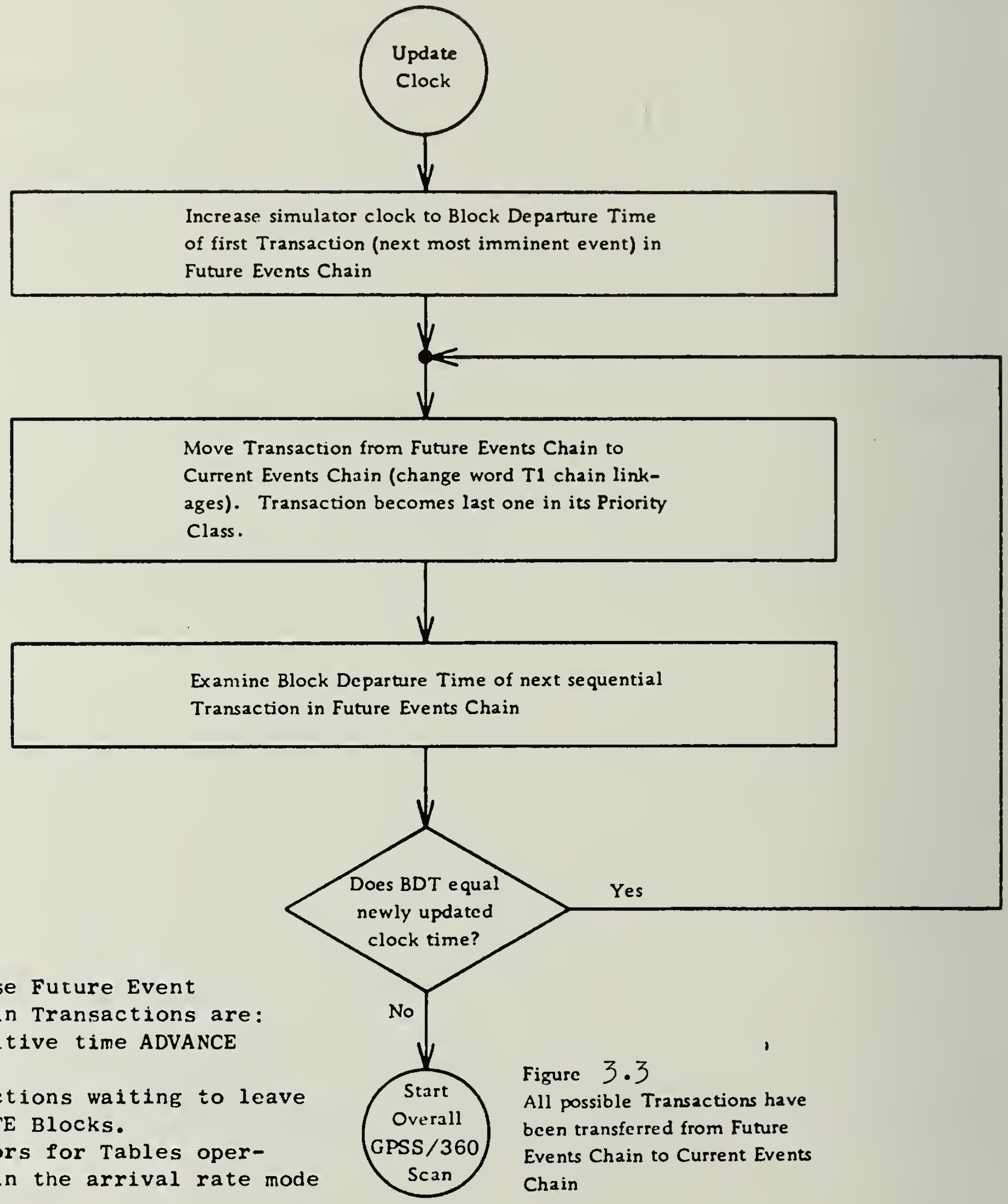
Selection of the transaction to move is based on two chains maintained by GPSS. The current events chain contains all transactions whose block departure times, BDT's, are equal to or less than the current time. Items in this chain are ordered first by priority then, within a priority class, on a first-in first-out basis. The future events chain contains all transactions whose BDT's are greater than the current time. For this chain, transactions are ordered by their BDT, most imminent event first, then for all transactions at a given time the ordering is as in the current events chain.

Transactions are selected for processing in order from the current events chain. Certain conditions which can prevent the movement of a transaction and therefore its processing are explained in following paragraphs. When processing of transactions on the current events chain is complete the first transaction in the future events chain, and all transactions with the same block departure time, are transferred to the current events chain.

The flowchart for the overall GPSS/360 scan is given in Figures 3.2, 3.3, and 3.4. The chart, from [7], is included to show the complexity of the algorithm and to provide a basis for comparison with the proposed machine organization. Figure 3.4 also gives the conditions under which the processing of one transaction is stopped and another is started.

Now the conditions mentioned above which can prevent the movement of transactions are discussed. A facility was described in section 3.2 as a unit capacity device. If a facility is in use at a time when another transaction would like to use (SEIZE) it, the second transaction must wait for the first to relinquish use. The second transaction has reached what is called a blocking condition. Processing is suspended until the facility becomes available, removing the blocking condition. A similar situation arises with respect to a storage when its capacity will be exceeded by the transaction that would like to use it. Thus when moving a transaction would violate the specifications of the system being simulated, the transaction is blocked.

Two blocks, GATE and TEST, can also stop the processing or divert the movement of a transaction. A gating condition can depend on the status

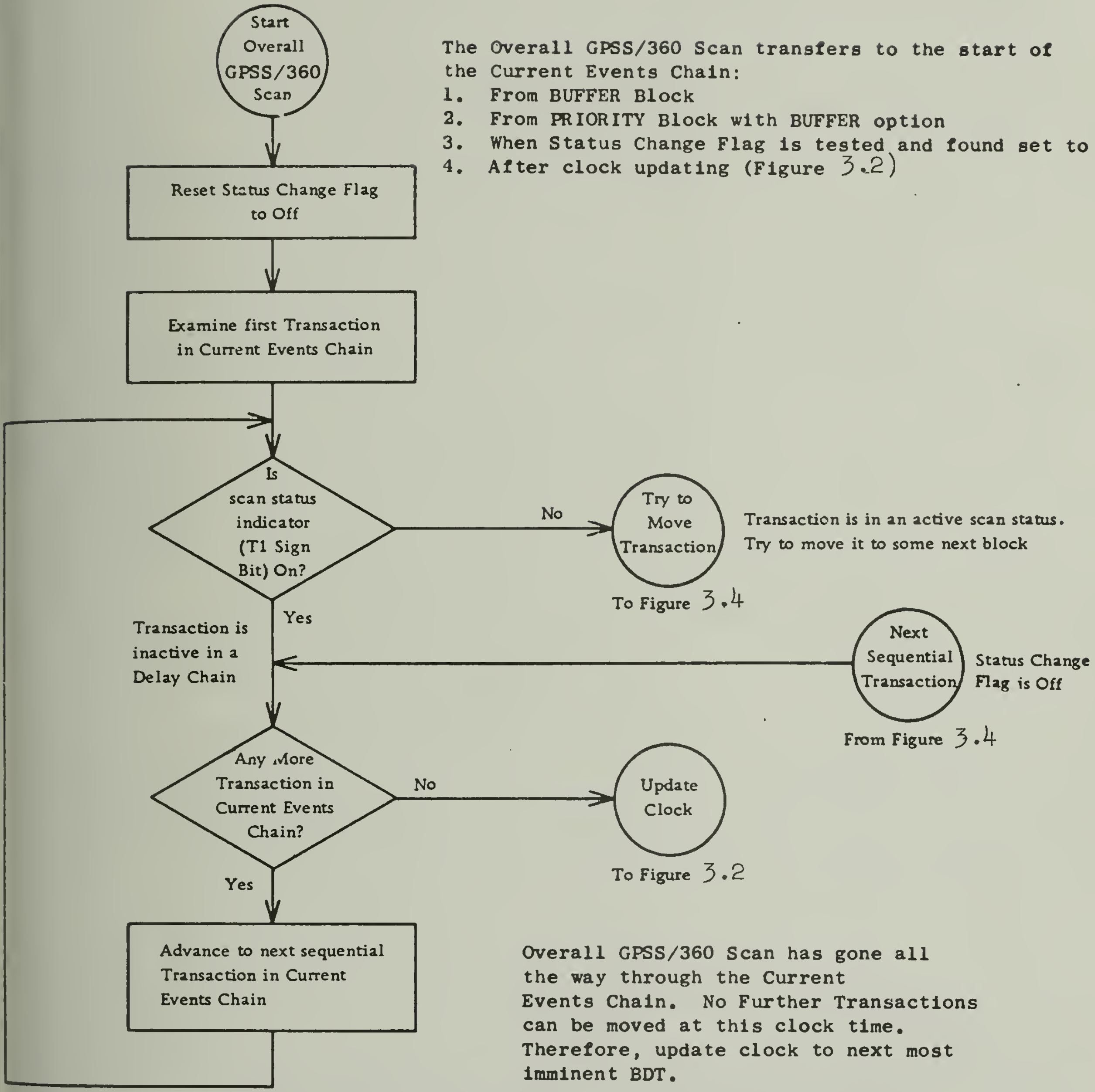


Update Clock
Increase simulator clock to Block Departure Time of first Transaction (next most imminent event) in Future Events Chain
Move Transaction from Future Events Chain to Current Events Chain (change word T1 chain linkages). Transaction becomes last one in its Priority Class.
Examine Block Departure Time of next sequential Transaction in Future Events Chain
Does BDT equal newly updated clock time?
Yes
No
Start Overall GPSS/360 Scan
Figure 3.3
All possible Transactions have been transferred from Future Events Chain to Current Events Chain

Note: These Future Event Chain Transactions are:

1. In positive time ADVANCE Blocks
2. Transactions waiting to leave GENERATE Blocks.
3. Operators for Tables operating in the arrival rate mode

Figure 3.2. Overall GPSS/360 Scan: Update Clock to Next Most Imminent Event

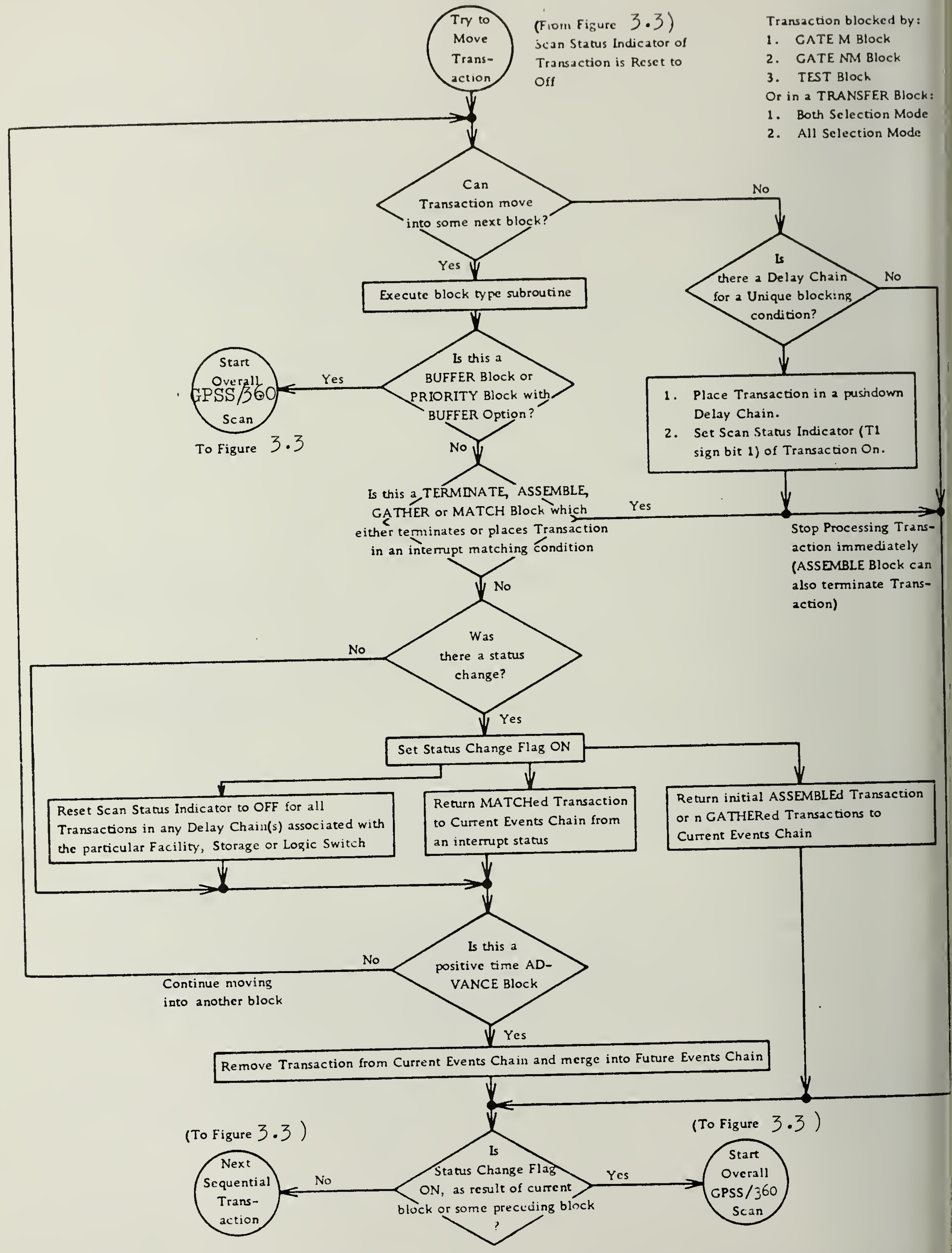

Start Overall GPSS/360 Scan
Reset Status Change Flag to Off
Examine first Transaction in Current Events Chain
Is scan status indicator (T1 Sign Bit) On?
No
Try to Move Transaction
Transaction is in an active scan status. Try to move it to some next block
To Figure 3.4
Yes
Transaction is inactive in a Delay Chain
Next Sequential Transaction
Status Change Flag is Off
From Figure 3.4
Any More Transaction in Current Events Chain?
No
Update Clock
To Figure 3.2
Yes
Advance to next sequential Transaction in Current Events Chain

The Overall GPSS/360 Scan transfers to the start of the Current Events Chain:

1. From BUFFER Block
2. From PRIORITY Block with BUFFER option
3. When Status Change Flag is tested and found set to on
4. After clock updating (Figure 3.2)

Overall GPSS/360 Scan has gone all the way through the Current Events Chain. No Further Transactions can be moved at this clock time. Therefore, update clock to next most imminent BDT.

Figure 3.3. Overall GPSS/360 Scan: Scan of Current Events Chain (Start of Scan)



Try to Move Transaction
(From Figure 3.3) Scan Status Indicator of Transaction is Reset to Off
Transaction blocked by:
1. GATE M Block
2. GATE NM Block
3. TEST Block
Or in a TRANSFER Block:
1. Both Selection Mode
2. All Selection Mode
Can Transaction move into some next block?
No
Yes
Is there a Delay Chain for a Unique blocking condition?
No
Execute block type subroutine
Is this a BUFFER Block or PRIORITY Block with BUFFER Option?
Yes
Start Overall GPSS/360 Scan
To Figure 3.3
1. Place Transaction in a pushdown Delay Chain.
2. Set Scan Status Indicator (T1 sign bit 1) of Transaction On.
No
Is this a TERMINATE, ASSEMBLE, GATHER or MATCH Block which either terminates or places Transaction in an interrupt matching condition
Yes
Stop Processing Transaction immediately (ASSEMBLE Block can also terminate Transaction)
No
Was there a status change?
No
Yes
Set Status Change Flag ON
Reset Scan Status Indicator to OFF for all Transactions in any Delay Chain(s) associated with the particular Facility, Storage or Logic Switch
Return MATCHed Transaction to Current Events Chain from an interrupt status
Return initial ASSEMBLEd Transaction or n GATHERed Transactions to Current Events Chain
Is this a positive time ADVANCE Block
No
Continue moving into another block
Yes
Remove Transaction from Current Events Chain and merge into Future Events Chain
(To Figure 3.3)
Next Sequential Transaction
No
Is Status Change Flag ON, as result of current block or some preceding block?
Yes
(To Figure 3.3)
Start Overall GPSS/360 Scan

Figure 3.4. Overall GPSS/360 Scan: Try to Move Individual Transaction into Some Next Block

of a logic switch, facility, storage, or block. The TEST block examines an algebraic relation between two standard numerical attributes. Thus a transaction can be blocked waiting for the model to satisfy certain programmer defined conditions.

When a transaction is blocked it depends on another transaction to change the model status and unblock it. In many cases a time change must take place before the model status changes. Time is a very important entity in control of the simulated system.

Serial execution of the program in Figure 3.1 is illustrated in Figure 3.5. The chart shows the sequence of transactions and blocks executed to complete the simulation. Processing starts at the bottom line. It progresses horizontally, moving the transaction that corresponds to the line through blocks until the processing must be stopped for one of the conditions given in Figure 3.4. Processing resumes with the transaction on the line above. Numbers on the lines are the times at which the transaction moved into the block. The final number on each line is the time when processing of that transaction was suspended, either temporarily or finally. The chart is derived from data gathered by tracing the execution of the example program.

There are several things to observe on the chart. Transactions one, two, and three move through all blocks without interruption. This is because no other transaction is active during the time they are in the model. There is no interaction between these three transactions so the movement of each through the program is equivalent to an execution of the program. Other transactions took two to four separate processing intervals to move through all blocks. Moving each of these through the program is like an execution of the program with interaction between the separate executions. Completion of the

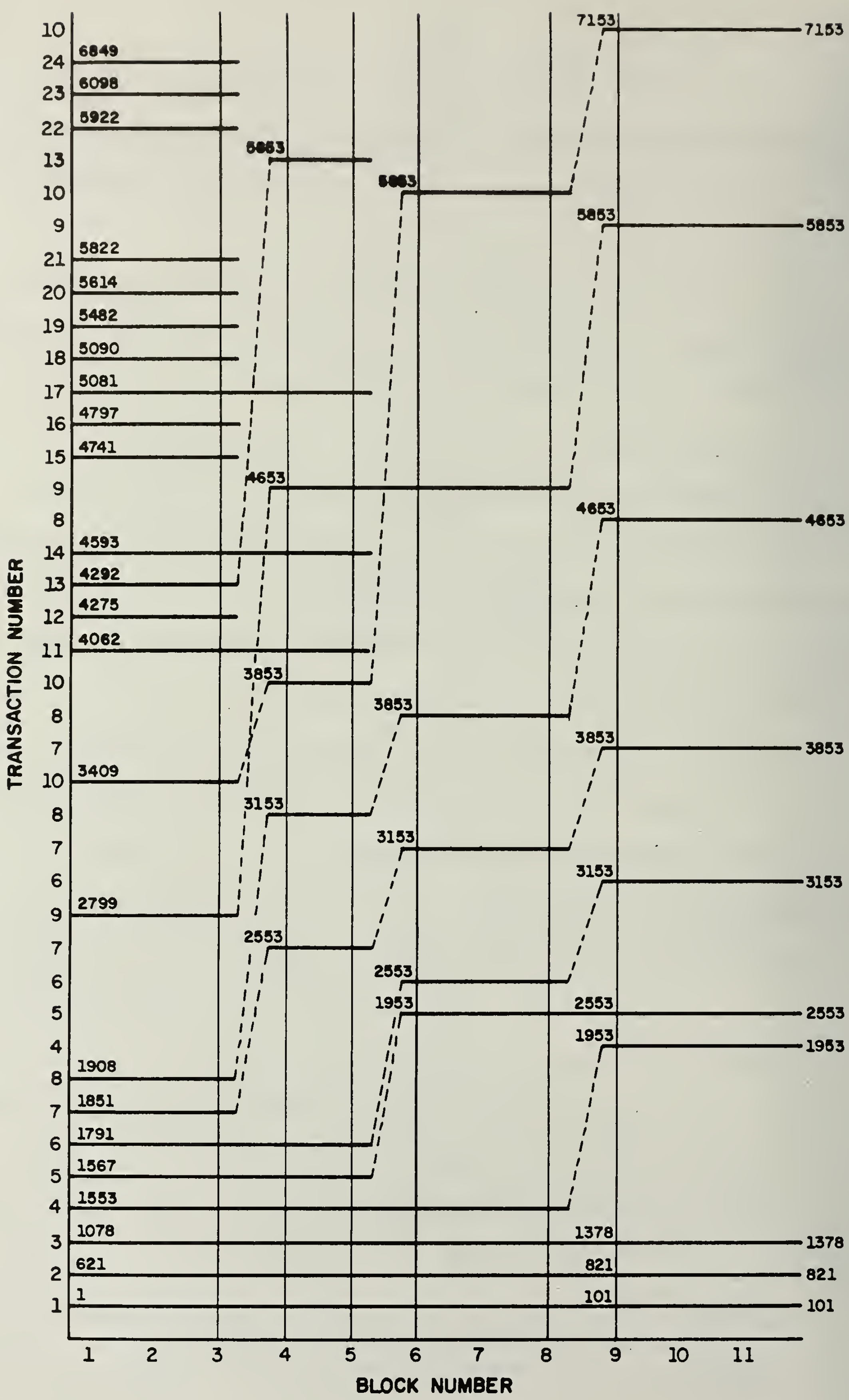


Figure 3.5. Serial Execution Trace of Example Program

simulation came with completion of ten transactions but more than ten started.

The order in which transactions are activated is not necessarily maintained throughout a simulation. An example of changing the order occurs on the chart. Transactions 10, 11, and 14 execute blocks four and five before transaction nine. Movement of transactions through the program is governed by run time determination of the model status. In general, the order of block executions is not known at compile time. Figure 3.5 demonstrates this. No amount of study of the program, short of actual execution, leads one to the knowledge that some transactions will move through the entire program with the processing of no other transaction intervening, whereas other transactions will require four distinct processing intervals.

3.3.2 Concurrent Execution

In this section three levels of parallelism within a GPSS program are described. The parallelism is used in the design of the multiprocessor machine of Chapter 4.

3.3.2.1 Parallelism Within a Block

The first level of parallelism is that which exists within the routine that a block type represents. This is precisely the type of parallelism that can be found in a procedural language. If it is known that a block is going to be executed, clearly the parallelism within the block can be fully used by a multiprocessor independent of consideration of other blocks or transactions. Use of this parallelism does not change the overall scan of Figures 3.2 through 3.4.

To measure the parallelism at this level 21 frequently used GPSS block types were converted from their original 360 Assembler Language version to Fortran. The 21 Fortran program equivalents were analyzed on the system described in [10]. Results of the analysis are given in Tables 3.1 and 3.2 at the end of this section.

Comments on the conversions to Fortran are given here. An attempt was made to have each Fortran program perform the same functions as the original although the methods had to differ slightly due to differences in the languages and operation of the analyzer program. The assembler language version makes use of many subroutines. In Fortran, subroutine calls were replaced by the subroutine itself so the analyzer could examine the complete program. Bits manipulated in assembler language were considered variables in Fortran. Specification statements were not given since they do not affect the parallelsim and are not examined by the analyzer.

Each block type is analyzed as a separate program so any exit from the block is an END or RETURN. This includes error checking statements which normally branch to the GPSS output phase, write an appropriate message, and terminate the simulation. In GPSS/360, completion of a block results in a call to the overall scan, Figure 3.4, and processing continues. For parallelism analysis each block is a separate program. No attempt is made to analyze sequences of blocks although this would tend to increase the parallelism.

Program conversion and analysis was not done for parts of the GPSS/360 system concerned with the selection of transaction-block pairs for execution. Only the box labeled "Execute block type subroutine" in Figure 3.4 of the overall scan algorithm was converted.

The flow chart and Fortran version listing of the QUEUE block are given in Figures 3.6 and 3.7. This block is a typical example of the type of statements and length of a block routine. Reference to these figures is made in Chapter 4 in a discussion of processor capability requirements.

Table 3.1 lists the results of analyzing the 21 block type programs, including QUEUE; to determine the number of operations that can be done concurrently and the speedup in execution using a multiprocessor configuration. The techniques and algorithms used in the analysis are fully described in [10].

One change was made to increase the accuracy of the analyzer for this set of programs. The analyzer previously did not count memory fetches under the assumption that they were overlapped with operations on the data. The listing of QUEUE shows very few arithmetic operations, reducing the overlap of memory fetches. Since other blocks are similar it is necessary to count fetches in the measurement of these programs.

The analyzer currently has a limited ability in handling IF trees. The maximum number of levels per tree is eight. An algorithm for folding trees with more levels has not been implemented yet so longer trees are artificially broken to give two or more trees. This results in less speedup than can be achieved with the hardware of Chapter 2.

Column headings for Table 3.1 are explained in the referenced paper and given again here. Minimum and maximum values are given when a range of results occurred.

(1) The names are the GPSS block type names, plus DECODE and FUNCTION, two routines for decoding operand values which required function evaluation or used indirect addressing for index values.

```
C          QUEUE BLOCK EXECUTION ROUTINE              NR 38000
      N =N+1
      IF (B3B6 .EQ. 1) GO TC  8400
      QUNITS =1
C                 QUENR SUB-ROUTINE                   NR  6000
 8101 IF (DCODA .LE. 0) GO TO 500
      IF (DCODA .GT. QUENUM) GO TO 500
C          END OF QUENR
C                 UPQ SUB-ROUTINE                     NR  7000
      IF (QNR .EQ. 0) GO TO 504
      IF (QNR .GT. QUENUM) GO TO 504
      IF (STIME .EQ. Q1)GO TO 7099
      DELTAT = STIME -Q1
      Q1 =STIME
      Q4 =Q4 +DELTAT *Q6
 7099 CONTINUE
C          END OF UPQ
      IF (T9B1 .NE. 1) GO TC  8100
      IF (T14B57 .EQ. 1) GO TO  8900
      QCOUNT =QCOUNT +1
      DO 8102 I =1,5
 8102 IF (MQTABL(I) .EQ. 0) GO TO  8105
      GO TO 669
 8105 MQTABL(I) =QNR
      MQTIME(I) =STIME
      GO TO 8300
 8100 IF (T13 .GT. 0) GO TO  8200
      T13 =QNR
      T14 =STIME
      GO TO  8300
 8200 T9B1 =1
      MQTABL(1) =T13
      MQTIME(1) =T14
      MQTABL(2) =QNR
      MQTIME(2) =STIME
      T14B6 =1
 8300 Q2 =Q2 +QUNITS
      Q6 =Q6 +QUNITS
      IF (Q6 .GT. Q7) Q7 =Q6
      RETURN
 8400 QUNITS =DCODB
      IF (QUNITS .NE. 0) GO TO  8101
      RETURN
 8900 IF (B3B1 .EQ. 1) GO TC  8300
      B3B1 =1
      CALL WRTMES      (A WARNING MESSAGE)
      GO TO  8300
  500 RETURN
  504 RETURN
  669 RETURN
      END
```

Figure 3.6. QUEUE Block; Fortran Version

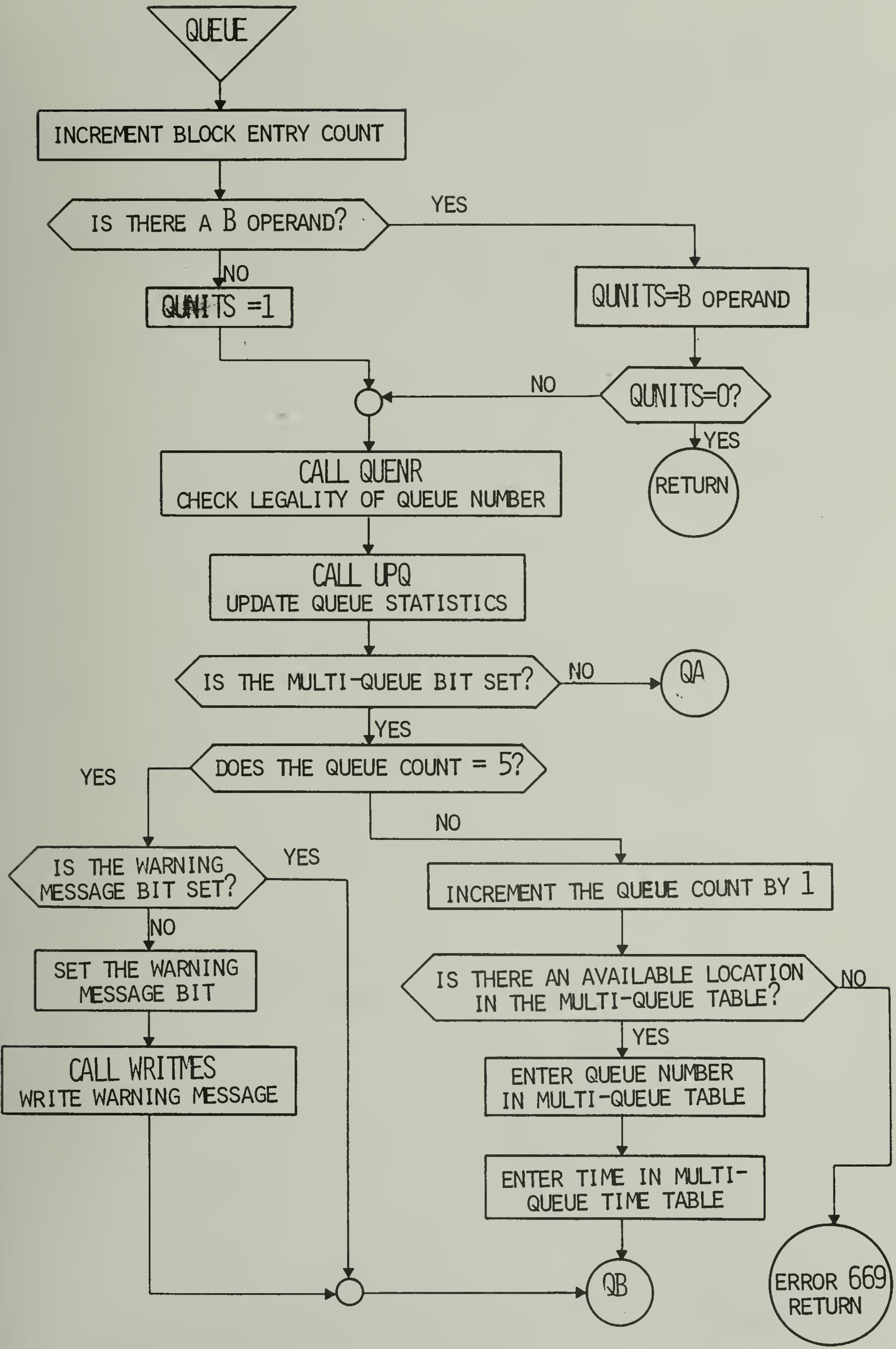

QUEUE
INCREMENT BLOCK ENTRY COUNT
IS THERE A B OPERAND?
YES
NO
QUNITS =1
QUNITS=B OPERAND
QUNITS=0?
NO
YES
RETURN
CALL QUENR
CHECK LEGALITY OF QUEUE NUMBER
CALL UPQ
UPDATE QUEUE STATISTICS
IS THE MULTI-QUEUE BIT SET?
NO
QA
YES
DOES THE QUEUE COUNT = 5?
YES
NO
IS THE WARNING MESSAGE BIT SET?
YES
NO
SET THE WARNING MESSAGE BIT
CALL WRITMES
WRITE WARNING MESSAGE
INCREMENT THE QUEUE COUNT BY 1
IS THERE AN AVAILABLE LOCATION IN THE MULTI-QUEUE TABLE?
NO
YES
ENTER QUEUE NUMBER IN MULTI-QUEUE TABLE
ENTER TIME IN MULTI-QUEUE TIME TABLE
QB
ERROR 669 RETURN

Figure 3.7. QUEUE Block Flow Chart

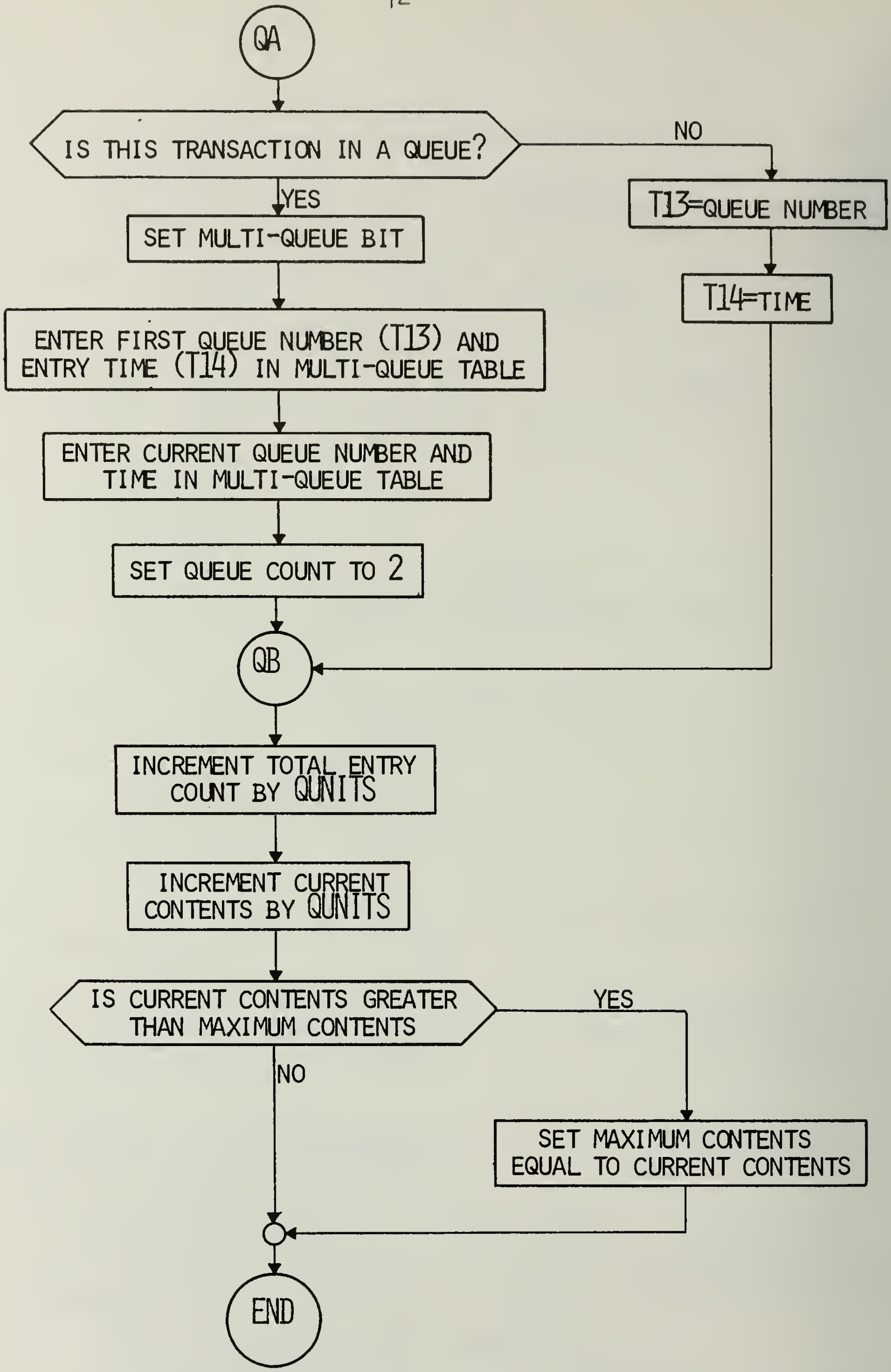

QA
IS THIS TRANSACTION IN A QUEUE?
NO
YES
SET MULTI-QUEUE BIT
ENTER FIRST QUEUE NUMBER (T13) AND ENTRY TIME (T14) IN MULTI-QUEUE TABLE
ENTER CURRENT QUEUE NUMBER AND TIME IN MULTI-QUEUE TABLE
SET QUEUE COUNT TO 2
T13=QUEUE NUMBER
T14=TIME
QB
INCREMENT TOTAL ENTRY COUNT BY QUNITS
INCREMENT CURRENT CONTENTS BY QUNITS
IS CURRENT CONTENTS GREATER THAN MAXIMUM CONTENTS
YES
NO
SET MAXIMUM CONTENTS EQUAL TO CURRENT CONTENTS
END

Figure 3.7 (continued). QUEUE Block Flow Chart

(2) This is the approximate number of source cards, excluding comments and multiple RETURN statements. The number of cards in the scopes of DO loops and IF loops is given.

(3) This is the maximum number of iterations assumed for any DO loop or IF loop in the program.

(4) The number of traces is the sum of all paths from the beginning to all END or RETURN statements plus the number of IF loops.

(5) T_1 is the time required to execute the program on a uniprocessor.

(6) T_p is the time required to execute the program using a multiprocessor capable of executing a maximum of p operations at once; a p-multiprocessor.

(7) S_p is the ratio of column (5) to column (6).

(8) This is the number, p, of processors required to achieve the T_p value in (6).

(9) E_p is the efficiency defined as:

$$E_p = \frac{T_1}{pT_p} \leq 1$$

for the number of processors in (8).

(10) U_p is the utilization for the number of processors in (8). The techniques used to reduce execution time may introduce extra operations. Let o_p be the number of operations in the execution of a program on a p-multiprocessor. Call R_p the operation redundancy and let

$$R_p = \frac{o_p}{o_1} \geq 1.$$

The p-multiprocessor utilization is $U_p = E_p R_p$.

(1) Name	(2) Number of Source Cards (#DO, #IF)	(3) Number of Iterations Assumed DO, IF	(4) Number of Traces Measured	(5) T_1	(6) T_p	(7) S_p	(8) p	(9) E_p	(10) U_p
ADVANCE	68(2,0)	8,-	70	24-176	7-35	1.90-6.25	11-12	.159- .583	.407- .660
ADVANCE*	25(0,0)	-	16	24-83	7-16	2.14-6.08	12	.178- .506	.392- .660
ASSIGN*	14(0,0)	-	6	16-26	9-17	1.25-2.00	2	.625-1.000	.718-1.000
DEPART	36(1,0)	5,-	73	14-94	7-24	1.71-5.86	5-26	.033- .685	.224- .742
ENTER	60(0,14)	-,1	153	14-146	7-55	1.06-6.29	7-13	.073- .612	.156- .653
GENERATE*	50(1,0)	8,-	41	8-162	7-28	1.14-6.24	8-13	.142- .678	.325- .678
INDEX	21(0,0)	-	8	14-61	9-11	1.56-5.55	7	.222- .792	.571- .792
LEAVE	56(0,21)	-,1	291	18-174	4-49	2.00-7.00	15-20	.082- .733	.148- .733
LINK	61(0,8)	-,4	97	8-226	8-50	1.00-7.20	12-23	.058- .321	.407- .639
LOGIC	32(0,14)	-,2	26	8-62	4-18	1.14-5.50	2-16	.056-1.000	.307-1.000
MARK	20(0,0)	-	8	10-51	7-9	1.11-5.67	8	.138- .708	.527- .708
MSAVEVALUE	34(0,0)	-	21	10-77	8-22	1.25-3.83	14	.142- .273	.298- .379
PRIORITY	6(0,0)	-	4	10-26	8-11	1.25-2.75	3	.416- .916	.575- .916
QUEUE	41(1,0)	5,-	53	14-90	14-29	1.00-5.00	10	.100- .500	.337- .500
RELEASE	30(0,7)	-,1	25	8-78	7-17	1.14-5.20	5-12	.228- .799	.380- .799
SAVEVALUE	28(0,0)	-	16	14-53	8-14	1.75-4.50	12	.145- .375	.428- .677

*Routine FUNCTION removed.

Table 3.1. Fortran Block Analysis

(1) Name	(2) Number of Source Cards (#DO, #IF)	(3) Number of Iterations Assumed DO, IF	(4) Number of Traces Measured	(5) T_1	(6) T_p	(7) S_p	(8) p	(9) E_p	(10) U_p
SEIZE	35(0,7)	-,1	20	8-72	7-18	1.14-5.43	9	.126- .603	.464- .619
SPLIT	79(52,0)	1,-	86	12-239	7-52	1.67-6.86	8-19	.155- .623	.298- .623
TERMINATE	29(0,2)	-,1	38	8-70	7-34	1.00-3.80	7	.122- .542	.338- .591
TEST	23(0,20)	-,1	21	28-40	8-11	2.91-6.50	4-21	.138- .226	.428- .553
TRANSFER	35(0,0)	-	23	18-59	7-19	2.29-7.86	18	.126- .436	.262- .603
UNLINK	105(0,47)	-,1	131	22-232	7-72	1.83-8.86	8-32	.182- .750	.335- .750
DECODE*	30(1,24)	1,1	119	40-114	9-42	1.68-6.56	4-12	.208- .527	.218- .527
FUNCTION	52(2,0)	8,-	28	8-142	8-21	1.00-7.05	11	.090- .710	.515- .710

*Routine FUNCTION removed.

Table 3.1 (continued). Fortran Block Analysis

Table 3.2 summarizes the program speedup information derived from the analysis. For each program it gives the number of traces with the speedup indicated in the column heading. Each column covers a range of ±0.5 from the heading value.

A conservative approach was taken in the design of the Fortran analyzer. The measures will change in the direction of improvement with improvement of the analyzer. These tables will be referenced in Chapter 4 on the design of a multiprocessor for executing these programs.

3.3.2.2 Concurrency Between Blocks

Since a single transaction moves through the blocks of a simulation program in much the same way as the execution of a conventional program proceeds from one instruction to the next, there is potential for concurrent execution of several blocks. This second level of parallelism is analogous to that between instructions in a procedural language but is at the level of routines rather than instructions.

This section is limited to simulations with one active transaction. The multiple transaction case is covered in the next two sections. If it is known that a sequence of blocks is going to be executed by one transaction, clearly the parallelism between blocks in the sequence can be exploited.

A GPSS program can be partitioned into groups of sequential blocks which do not violate the precedence requirements of variables. That is, from the definition of each block type and examination of its operands it is possible to group blocks such that no variable is written then read by the blocks in one group.

Block Type	Speedup Factor									Modal
	1	2	3	4	5	6	7	8	9	Values
ADVANCE*		3	4	3	4	2				3,5
ASSIGN*	1	5								2
DEPART		6	11	14	28	14				5
ENTER	4	42	77	15	1	14				3
GENERATE*	1	1	2	10	16	11				5
INDEX		2	-	3	2	1				4
LEAVE	1	33	12	103	85	51	6			4
LINK	1	1	1	22	36	29	7			5
LOGIC	1	5	12	4	-	4				3
MARK	1	1	1	2	2	1				4,5
MSAVEVALUE	1	5	9	6						3
PRIORITY	1	2	1							2
QUEUE	3	9	18	20	3					4
RELEASE	1	1	7	11	3	2				4
SAVEVALUE		2	6	6	2					3,4
SEIZE	1	1	4	8	4	2				4
SPLIT		4	7	21	36	17	1			5
TERMINATE	3	16	18	1						3
TEST			3	11	4	1	2			4
TRANSFER		1	11	2	3	3	2	1		3
UNLINK	6	50	43	8	5	2	7	3	3	2
DECODE*	7	18	42	26	16	9	1			3
FUNCTION	1	2	1	1	6	10	6	1		6

*FUNCTION subroutine removed for analysis.

Table 3.2. Block Routine Speedup Factors

For a simulation with one transaction the blocks in a precedence partition are those that can be processed concurrently. Most simulations involve multiple transactions which affect each other and the system being modeled. As a result, the processing of one transaction may be interrupted at a block in the midst of a precedence partition. The reduction of concurrency is covered in section 3.3.3.

A segment of the program in Figure 3.1 is reproduced here to serve as a precedence partitioning example. Selected assignment statements from the corresponding routines are given at the right.

Block Number	Block	Operands	Selected Routine Statements
2	ASSIGN	12,FN$TERMI	P12 $\leftarrow$ Function value
3	INDEX	3,10	
4	SEIZE	P12	J $\leftarrow$ P12 F(J) $\leftarrow$ 1

P12 appears on the left side of an assignment statement in block two and on the right in block four. Since block four cannot be executed concurrently with two, they must be in separate partitions.

The order of block execution is known for all segments of the program free of conditional jumps. If interaction between transactions is not considered, all blocks in a partition can be processed concurrently. The parallelism within each block can be simultaneously applied. The overall scan in Figures 3.2 through 3.4 is changed to the extent that all blocks in a partition are executed concurrently rather than serially.

Algorithm 3.1 for the precedence partitioning of GPSS programs is presented in Figure 3.8. Basically, the GPSS card deck is scanned sequentially.

When a block is found, the type and standard numerical attributes, SNA's, used as operands are noted. If processing the block simultaneously with preceeding blocks in the current partition would violate precedence requirements, the current partition is closed and the block becomes the first in a new partition.

Due to the structured nature of GPSS it is not necessary to examine the statements of each routine to perform the partitioning. The algorithm uses the Precedence Partitioning Guide, Table 3.3, which is based upon the definition of each block type, to identify the blocks or SNA's that cannot be in the same partition as the current block.

Entries in column B of the guide are those blocks which read a variable written by the block type in the corresponding position of the left most column. When the entry for a block type is ALL, the block type is the last in a partition regardless of what follows. Block types EXECUTE and GENERATE are permanent entries in column B for all block types. Optional block operands cause an EXCLUSIVE OR situation for some entries. The guide is conservative in several respects. The TRANSFER block causes a partition boundary even when it is used for unconditional jumps which do not violate any operand precedence requirements. Block types which are used in pairs, such as QUEUE and DEPART, are not permitted in the same partition. The actual requirement is that pair types should not be in the same partition if they are operating on entities with the same index value.

Entries in column S are those SNA's which are given new values by the block type in the left column. In several cases a single letter is used to represent a set of SNA's. This is the case for example with ENTER where S is used for the set S, SR, SA, SM, SC, and ST. The ENTER block can change values

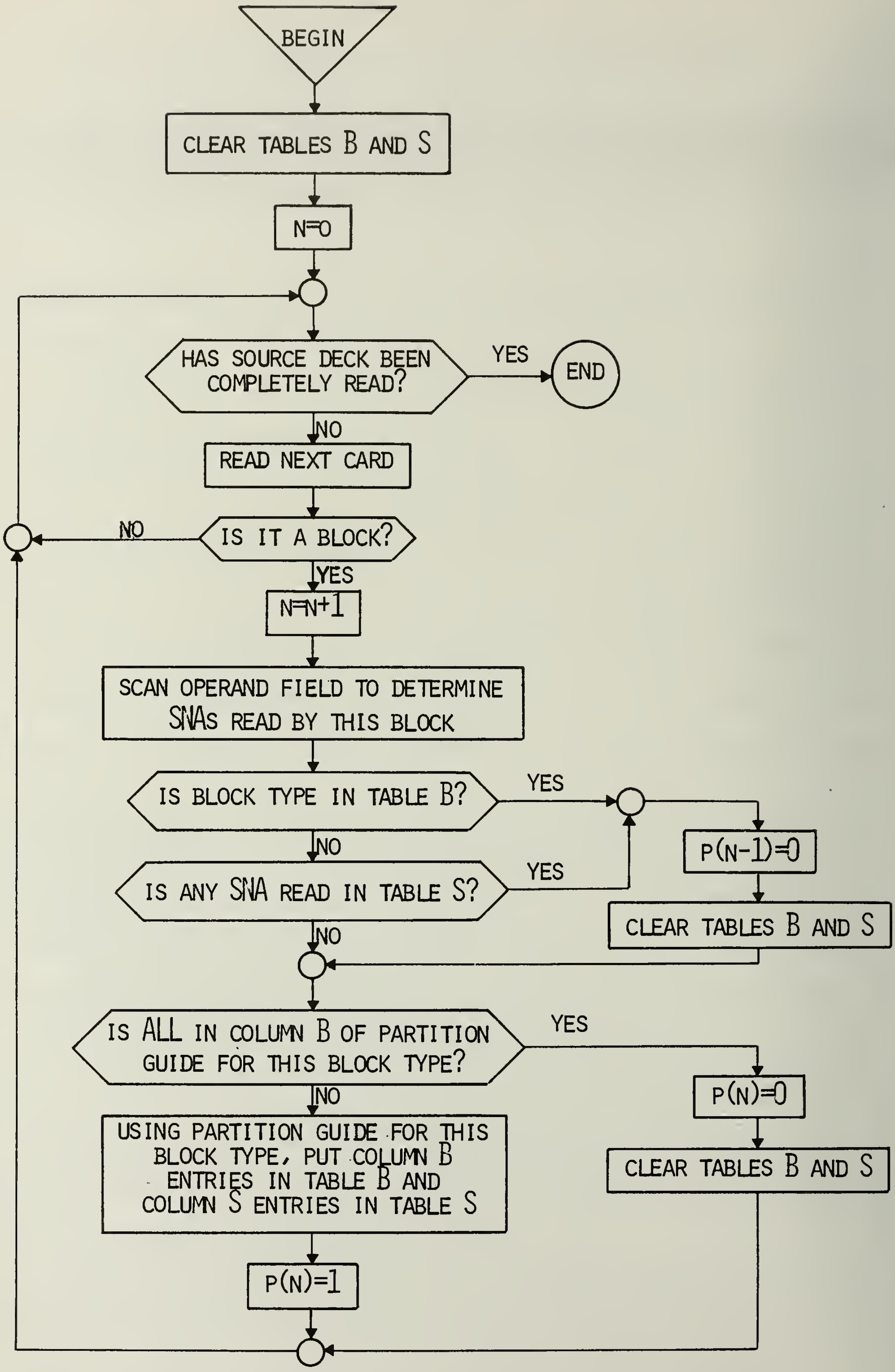

BEGIN
CLEAR TABLES B AND S
N=0
HAS SOURCE DECK BEEN COMPLETELY READ?
YES
END
NO
READ NEXT CARD
IS IT A BLOCK?
NO
YES
N=N+1
SCAN OPERAND FIELD TO DETERMINE SNAs READ BY THIS BLOCK
IS BLOCK TYPE IN TABLE B?
YES
NO
IS ANY SNA READ IN TABLE S?
YES
P(N-1)=0
CLEAR TABLES B AND S
NO
IS ALL IN COLUMN B OF PARTITION GUIDE FOR THIS BLOCK TYPE?
YES
P(N)=0
CLEAR TABLES B AND S
NO
USING PARTITION GUIDE FOR THIS BLOCK TYPE, PUT COLUMN B ENTRIES IN TABLE B AND COLUMN S ENTRIES IN TABLE S
P(N)=1

Figure 3.8. Precedence Partition Algorithm

Precedence Partitioning Guide

Block Type	Precedence Table Entries	
	B	S
	EXECUTE, GENERATE	
ADVANCE	ADVANCE, ENTER, GATE, LEAVE, LOGIC, MARK, PREEMPT, RELEASE, RETURN, SEIZE	C1, MP, M1
ALTER	(EXAMINE, JOIN, REMOVE, SCAN) $\oplus$ ALL if operand G is specified	PR $\oplus$ Pj, j = operand C
ASSEMBLE	ALL	
ASSIGN		Pj, j = operand A
BUFFER	ALL	
CHANGE	ALL	
COUNT		Pj, j = operand A
DEPART	QUEUE	Qj, j = operand A
ENTER	LEAVE	Sj, Rj, j = operand A
EXAMINE	ALL	
EXECUTE	ALL	
GATE	ALL	
GATHER	ALL	
GENERATE	ADVANCE, ENTER, GATE, LOGIC, MARK, PREEMPT, SEIZE	
INDEX		P1
JOIN	ALTER, EXAMINE, REMOVE, SCAN	Gj, j = operand A
LEAVE		Sj, Rj, j = operand A

Table 3.3. Precedence Partitioning Guide

Precedence Partitioning Guide

Block Type	Precedence Table Entries	
	B	S
	EXECUTE, GENERATE	
LINK	ALL	
LOGIC		Lj, j = operand A
LOOP	ALL	
MARK		(MPj, Pj, j = operand A) $\oplus$ M1
MATCH	ALL	
MSAVEVALUE		MXj(k,1) $\oplus$ MHj(k,1) j = operand A k = operand B 1 = operand C
PREEMPT	RETURN	Fj, j = operand A
PRINT		
PRIORITY	ALL if operand F is specified	PR
QUEUE	DEPART	Qj, j = operand A
RELEASE		Fj, j = operand A
REMOVE	(ALTER, EXAMINE, JOIN, SCAN) $\oplus$ ALL if operand F is specified	Gj, j = operand A
RETURN		Fj, j = operand A
SAVEVALUE		Xj $\oplus$ XHj, j = operand A
SCAN	(ALTER, JOIN, REMOVE) $\oplus$ ALL if operand F is specified	Pj, j = operand E
SEIZE	RELEASE	Fj, j = operand A
SELECT		Pj, j = operand A

Table 3.3 (continued). Precedence Partitioning Guide

Precedence Partitioning Guide

Block Type	Precedence Table Entries	
	B EXECUTE, GENERATE	S
SPLIT		Pj, j = operand C
TABULATE		Tj, j = operand A
TERMINATE	ALL	
TEST	ALL	
TRACE	ALL	
TRANSFER	ALL	
UNLINK		Cj, j = operand A
UNTRACE		

Table 3.3 (continued). Precedence Partitioning Guide

for each set member so the use of any member causes a partition boundary and detection of S is sufficient.

The WRITE block is omitted from the guide since it involves tape features which are not being considered.

Block precedence partition membership is indicated by a subscripted variable P(n) where n is the block number assigned in order of appearance in the source deck. P(n) = 1 when the following block belongs to the same partition. P(n) = 0 for the last block in a partition.

Applying the algorithm to the example program gives the result in Figure 3.9. Simulation of one transaction moving through this program can be done in four processing "intervals", corresponding to the four partitions, on a suitable multiprocessor.

Block Number	Block	Operands	P(n)	Partition
1	GENERATE	300,FN$EXPON	1	1
2	ASSIGN	12,FN$TERMI	1	1
3	INDEX	3,10	0	1
4	SEIZE	P12	1	2
5	QUEUE	CPU	1	2
6	SEIZE	CPU	0	2
7	DEPART	CPU	1	3
8	ADVANCE	V1	0	3
9	RELEASE	CPU	1	4
10	RELEASE	P12	1	4
11	TERMINATE	1	0	4

Figure 3.9. Precedence Partitions of Program in Figure 3.1

3.3.2.3 Concurrent Transaction Movement

The interesting feature of simulation programs is the prospect for concurrently processing more than one transaction. This third level of parallelism is without analogue in conventional programs.

In section 3.3.1 the point was made that a single transaction moving through a simulation program was like a single execution of a conventional program. If multiple transactions moving through a simulation program were like multiple executions of a conventional program it would be a straightforward matter, in theory, to concurrently process as many transactions as desired. This is not, however, the case. There is an interrelationship of transactions with the model, including other transactions, that must be taken into account.

It has been mentioned, in section 3.3.1, that the processing of a transaction can be interrupted for several reasons and that transactions can move through the simulation program at different rates causing changes in their ordering in the program. These effects come about because all transactions are related to the model and each transaction can affect variables to which other transactions have access. A <u>system variable</u> is a variable which can be both written and read by more than one transaction. Any block which uses a system variable as an operand that will be read must be processed in a certain order with respect to blocks that write that variable. This means that precedence requirements extend across transactions and become a run time function rather than being defined by the program at compile time.

A restricted application of multiple transaction concurrency would be to select for processing only the set of transactions which represent current events. This set consists of the members of the current events chain described in the serial execution section, 3.3.1. The restriction is not necessary if proper care is exercised in the selection of transactions to move.

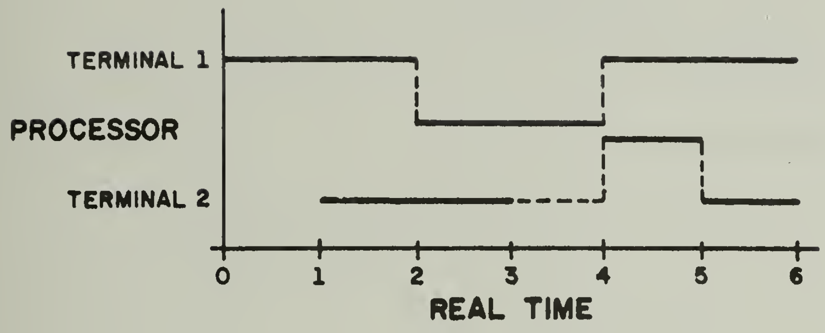
That is, members of the future events chain are also candidates for concurrent processing.

The essence of this is that a range of the real time being simulated can be involved in processing being done concurrently. At an instant of real time on a multiprocessor, more than one instant of simulated time can be in process. All of the interactions that take place in real time in the model must be considered, plus the interactions caused by overlapping multiple instances of simulated real time into a single instant on the computer.

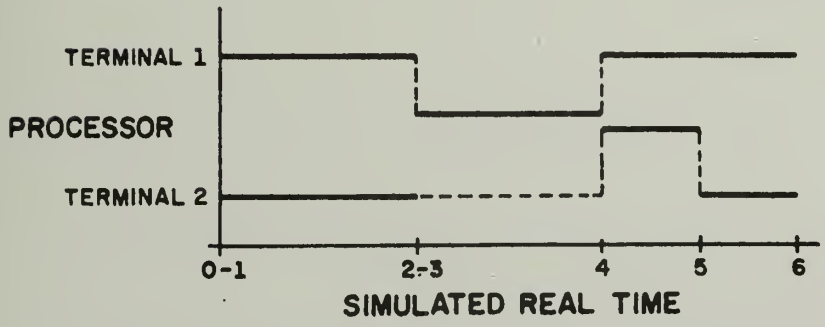
As an example of time overlap consider a computer system with two independent terminals and one processor that can be called into exclusive use by either terminal. If the processor is in use when a terminal requests it, the terminal must wait for the processor to become available. Figure 3.10(a) shows a possible use of the equipment on an arbitrary real time scale. Solid lines represent equipment in use, dashed lines represent time spent waiting for the processor, and vertical dashes are transitions between equipment.

Figure 3.10(b) shows the same equipment usage as it can be simulated. An event at real time one can be processed simultaneously with an event at real time zero if the events are independent, as they are in this example. Likewise for events at real times two and three.

In Figure (a) there are six distinct times at which events to be simulated take place. They are tabulated below. On a multiprocessor restricted to simulating all events at one instant of time before advancing to the next, six distinct processing intervals would be required. In Figure (b) with the overlap of times (0,1) and (2,3) there are only four distinct times. On a multiprocessor which allows such overlap only four distinct processing intervals are required.



(a) Timing of System to be Simulated



(b) Timing with Overlap

Figure 3.10. Example of Time Overlap on Independent Events

Real Time	Events	Processing Interval
0	Initiate Terminal 1	1
1	Initiate Terminal 2	2
2	Terminal 1 takes Processor	3
3	Terminal 2 requests Processor	4
4	Terminal 1 frees and Terminal 2 takes Processor	5
5	Terminal 2 frees Processor	6

(a) Processing of Figure 3.10(a)

Simulated Real Time	Events	Processing Interval
(0,1)	Initiate Terminals 1 and 2	1
(2,3)	Terminal 1 takes and Terminal 2 requests Processor	2
4	Terminal 1 frees and Terminal 2 takes Processor	3
5	Terminal 2 frees Processor	4

(b) Processing of Figure 3.10(b)

Table 3.4. Distinct Events Comparison of Figure 3.10

For this example, the time overlap allowance results in a savings of two processing intervals. The error that must be avoided however is letting terminal two take the processor before terminal one since they both appear ready at the same place on the scale in Figure 3.10(b). That is an example of the interaction introduced by overlapping multiple instances of simulated real time.

A specific GPSS example of the precedence requirements due to multiple transactions is now taken from the program in Figure 3.1. Again, a program segment is reproduced. The single transaction precedence partition numbers are listed as determined in the previous section.

Block Number	Block	Operands	Selected Routine Instructions	Partition
4	SEIZE	P12	N4 = N4+1	2
5	QUEUE	CPU		2
6	SEIZE	CPU		2
7	DEPART	CPU		3
8	ADVANCE	V1	TIME = TIME+100*N4	3
	1 VARIABLE	K100*N4		

Blocks four and eight are in separate partitions and therefore will not be processed concurrently for any single transaction. The writing and reading of N4 appears to be properly separated. Consider the following multiple transaction situation which uses times that actually occurred as taken from Figure 3.5.

Suppose transaction six is processing partition two including block four at time 1791 and, concurrently, transaction five is processing partition three including block eight at time 1953. Each transaction is following the precedence partition but six is writing N4 while five is reading it.

Even though transaction five has executed block four it cannot correctly execute block eight until all transactions with times less than 1953 have moved through block four. It is therefore not sufficient just to follow the precedence that can be detected at compile time.

A similar situation does not exist with operand P12 which occurs in three different partitions. P12 refers to parameter 12 of the transaction executing the block. In this program no transaction can change the P12 value of another.

Fortunately the variables and block types involved in precedence requirements between transactions are known. The variables have been previously defined in this section as system variables. There is a special category of three standard numerical attributes--function, variable, and boolean variable--which are programmer defined. They are system variables if the definition uses a system variable. The system block types are those which, by definition, implicitly use system variables.

The use of system variables is indicated for each block by S(n) where n is the block number, as in the precedence partition algorithm. S(n) = 1 when the block does not use any system variables due to either the block type or the operands. S(n) = 0 when it does. The assignment of values to S(n) is made on a block by block basis simply by comparing the type and operands against a table of system blocks and variables.

Allowing concurrent processing of multiple transactions changes the concept of the overall scan, Figures 3.2 through 3.4, drastically. The current and future events chains are eliminated. A single clock is replaced by a time word for each transaction. The serial nature of the algorithm must be revised to take advantage of the parallelism in the program. In Chapter 4 a hardware unit is proposed as a replacement for the current software overall scan algorithm.

Elimination of the two chains makes it necessary to define some new terms. Transaction time is the simulated time word associated with each transaction. It is similar to and replaces, the block departure time. A min time transaction is one whose transaction time is equal to or less than the transaction time of all other transactions which are not blocked. Refer to section 3.3.1 for the meaning of blocked. The set of min time transactions is the set of transactions which would have been members of the current events chain.

The significance of S(n), mentioned above, is that any block with S(n) = 0 must be processed only by a min time transaction. The effect, referring to the example in Figure 3.10, is that the action of taking the processor can be restricted to the min time transaction. This assures terminal one of success in taking the processor at time two, avoiding the potential error mentioned in the example. Note that if S(n) = 0 for all blocks, processing reverts to the time ordered case. S(n) = 1 allows those blocks which are time independent to be processed by non-min time transactions.

3.3.3 Processing Code Assignment

In the next chapter a multiprocessor machine organization for processing simulations in GPSS is presented. The machine requires information on precedence partitions and system variable usage to control the processing. Slight variations on the P(n) and S(n) assignments of the previous two sections are needed however. The modified values are then concatenated to form a two bit code, SP(n), for each block. This section discusses the modifications and the final code.

First the P(n) changes are discussed. The three blocks SEIZE, PREEMPT, and ENTER which have the action of letting a transaction use a facility or storage are assigned P = 0. In the multiple transaction case it is possible for the equipment to be in use such that the transaction is blocked preventing the execution of the next block. Letting P = 0 creates an artificial precedence boundary which prevents execution of the next block until the previous partition is successfully concluded. This was not necessary in the single transaction case since a blocked transaction would be a deadlock. No other transaction would ever be available to clear the blocking condition. The program itself is faulty if this happens.

Simulations with multiple transactions do cause a reduction in the length of precedence partitions and thus a decrease in the number of concurrently executable blocks per transaction. The total concurrency increases however, due to more than one transaction being processed.

The BUFFER block is used to stop the movement of a transaction when it could normally move to the next block. The effect of this is achieved by assigning P = 0 to the previous block and removing the BUFFER. PRIORITY with the BUFFER option is assigned P = 0 for the same reason.

Normally GATE and TEST blocks, which are like conditional jumps for transactions, receive P = 0 since the next block is unknown. The entry ALL in column B of the partitioning guide causes this. When GATE or TEST blocks with alternate exits appear in a sequence, an IF tree for transactions is formed. Appropriate revision of the routines for these two block types will allow combined execution of the sequence as one block of greater length, making effective use of the hardware in Chapter 2. When the sequence occurs the precedence partitioning algorithm is applied to all blocks, except the last, as if the column B entry in the guide were blank. Thus operand precedence is checked to determine P rather than assigning P = 0 directly.

The above modifications change the partitioning algorithm from defining single transaction precedence partition boundaries to defining sets of blocks which are simultaneously executable in a multiple transaction environment. In the remainder of this paper the sets will be referred to as partitions.

Now consider the modifications to S(n). The purpose of this bit is to flag blocks which must be executed only by min time transactions. If two such blocks are in sequence with no possible change in transaction time from one to the other, it is only necessary to flag the first one. If the transaction is at the minimum time on execution of the first block it will necessarily be there for the second also, regardless of the value of S. Therefore assign the second block S = 1. The choice is made to improve performance in the machine and will be explained in section 4.4.

This asssignment rule can be extended to allow any number of blocks in the sequence and also allow intervening blocks which do not use system variables. One condition is necessary however. If a labeled block, or other block which can be the destination of a jump, occurs, it breaks the sequence. A transaction which is not at the minimum time can transfer to such a block and must be prevented from executing a system variable block.

A program has been written that scans GPSS decks, simultaneously applying the partition algorithm and system variable table look-up, with modifications mentioned above, to generate the code bits SP(n) for each block. The result is an indication of the concurrency in the program but is of course a static measure. It does not measure the effects of multiple transactions or selected paths through the program.

Results of scanning several programs are given in Table 3.5. The meanings of the column headings are:

(1) The name identifies the application of the program. The source of the listing is given by reference.

(2) Blocks are the executable GPSS statements.

(3) Blocks which unconditionally transfer transactions are tabulated. They shorten partitions since they are considered the last block in a partition without inspecting the destination of the transfer.

(4) The maximum partition length is the largest number of blocks that are simultaneously executable.

(5) The average partition length is the average number of simultaneously executable blocks.

(6) This column is the number of blocks for which S = 1. The processing of these blocks is never delayed to wait for min time transactions.

(7) This column is the number of blocks for which P = 1.

(8) This column is the number of blocks for which both S and P are one.

The maximum and average partition lengths are the figures of most interest. For all the programs that were scanned the average is near two. Thus, on the average, any transaction at any time can be executing two blocks concurrently.

It is felt that the average could be raised to three blocks with a more sophisticated scan program. The current program implements the algorithm of Figure 3.8 with the modifications of this section. This algorithm does a sequential scan of the source deck. Improvements could be made by following traces through the program so transaction decision trees could be found and the destination of unconditional transfers could be examined. Loops made with the LOOP block could be expanded. This has been done by hand for two

(1) Program Name	(2) Number of Blocks	(3) TRANSFERs		(4) Maximum Partition Length	(5) Average Partition Length	(6) Number of Blocks with S=1	(7) Number of Blocks with P=1	(8) Number of Blocks with SP=1
		Number	%					
Control Unit (Appendix)	181	15	8	8	2.06	144	93	83
Memory [12]	175	15	9	10	2.22	141	96	84
Job Shop [14]								
(1) As is	80	4	5	6	1.95	61	39	27
(2) Loops expanded	88	4	5	9	2.26	69	49	37
Elevator [1]	53	5	10	5	2.30	46	30	25
Rail Fleet [6]	49	5	10	5	1.81	38	22	17
Thesis Example (Figure 3.1)	11	0	0	3	2.20	8	6	5

Table 3.5. Program Partitions and Processing Codes

of the four loops in the Job Shop program with a resulting improvement of .31 blocks on the average partition length. The total results are listed as Job Shop (2). Techniques such as re-ordering blocks where the order is not critical, and back substitution of variables across blocks could also be used to increase partition lengths.

In Chapter 4 the use of this code for coordinating a multiprocessor is described. The machine design allows the three levels of parallelism discussed in this chapter. It is recognized that implementation of GPSS on the multiprocessor will require some revision of the block routines to account for the change from time ordered to time overlapped processing.

4. MACHINE DESIGN FOR CONCURRENT EXECUTION OF DISCRETE TIME SIMULATIONS

4.1 Introduction

A simulation language, GPSS, has been examined for the purpose of finding ways to speed up the execution. The speedup was intended to come from multiprocessing of the existing language, not from changing the language definition. Results of the examination revealed parallelism within routines that make up the language and parallelism between those routines. The parallelism will be used in this chapter to design a multiprocessor machine for concurrent execution of simulation studies.

The largest readily identified part of a GPSS simulation which is known to be executed in its entirety is a block, the GPSS name for the routines of the language. A specific routine is executed when a transaction moves into a block with the name of the routine. For example, in Figure 3.1 the routine for the SEIZE block is executed each time a transaction moves into block four, and again when it moves into block six. The operands for these two blocks are not the same. Operands are parameters which identify data and execution options for the routine. The combination of a transaction, a block type, and a set of operands defines a routine to be executed and the data for this execution. The block type and operands are simply a particular program block. The pair consisting of a transaction and a particular program block is called a task.

In section 3.3.2 three levels of parallelism in a GPSS program were described. The first level is that within a block, or in execution terminology, within a task. The second level establishes the fact that tasks for a given transaction can be simultaneously executable. The tasks which are simultaneously executable constitute a static set determined at compile time. The third level extends the concept of sets of simultaneously executable tasks to include tasks related to more than one transaction. In this case the sets become dynamic; their members are functions of the run time status of the model.

Suppose a multiprocessor machine were designed to determine a set of tasks that could be processed simultaneously, process them, then request the next set. There are two weaknesses with this scheme. First, all tasks in the set do not take the same amount of processor time for execution. If all processors are forced to wait for the longest task, all processors except one will be idle for some time after completion of their tasks. The second weakness is that the number of tasks in a set will not, in general, fit the number of processors an integral number of times. A set with an excess of one task results in total idleness for all processors except one and delays execution of the next set.

A better scheme would be to keep a list of tasks which are ready for execution and let processors operate asynchronously. When a processor completes a task it is available for the next task in the list. This eliminates the two weaknesses caused by differences in task execution times and the misfit between the number of members in a set and the number of processors. The list of executable tasks is the basis of the machine organization presented here.

GPSS/360 uses a software overall scan algorithm, Figures 3.2 through 3.4, to serially select the next task. That algorithm is replaced by a hardware coordination unit that selects transactions to move, and builds the list of tasks ready for execution. The coordination unit is thus the source of tasks for distribution to a group of processors.

An overall picture of the machine organization is given in Figure 4.1. Explanation of the figure and details concerning the processors, coordination unit, and memories will be given in this chapter. The details are partially based on the results presented in Chapter 3. Since all major parts of the machine work together to some extent, the discussion of any one part is incomplete without reference to, or knowledge of, the other parts. The discussion, therefore, is of necessity somewhat circular and incomplete until all parts have been covered.

4.2 Compilation Observations

When a person uses GPSS to perform a simulation study he writes a "program" in which the executable statements are GPSS blocks. Recall that block types are routines provided by GPSS for the user, not written by the user. The "program" is a sequence of calls for the execution of previously written routines.

It is only necessary to compile the fixed set of routines one time for all users. Compiled routines can be stored in the machine in a read only memory. Since compilation is a one time happening it is not unreasonable to tune the code produced to the machine. Optimization can consider the interconnection of processors and memories to minimize memory access conflicts. Detection of IF trees and use of the algorithms of Chapter 2 for preparation of trees, and their mapping into the decision processor, is included in compilation.

Existence of compiled code for the block routines does not mean

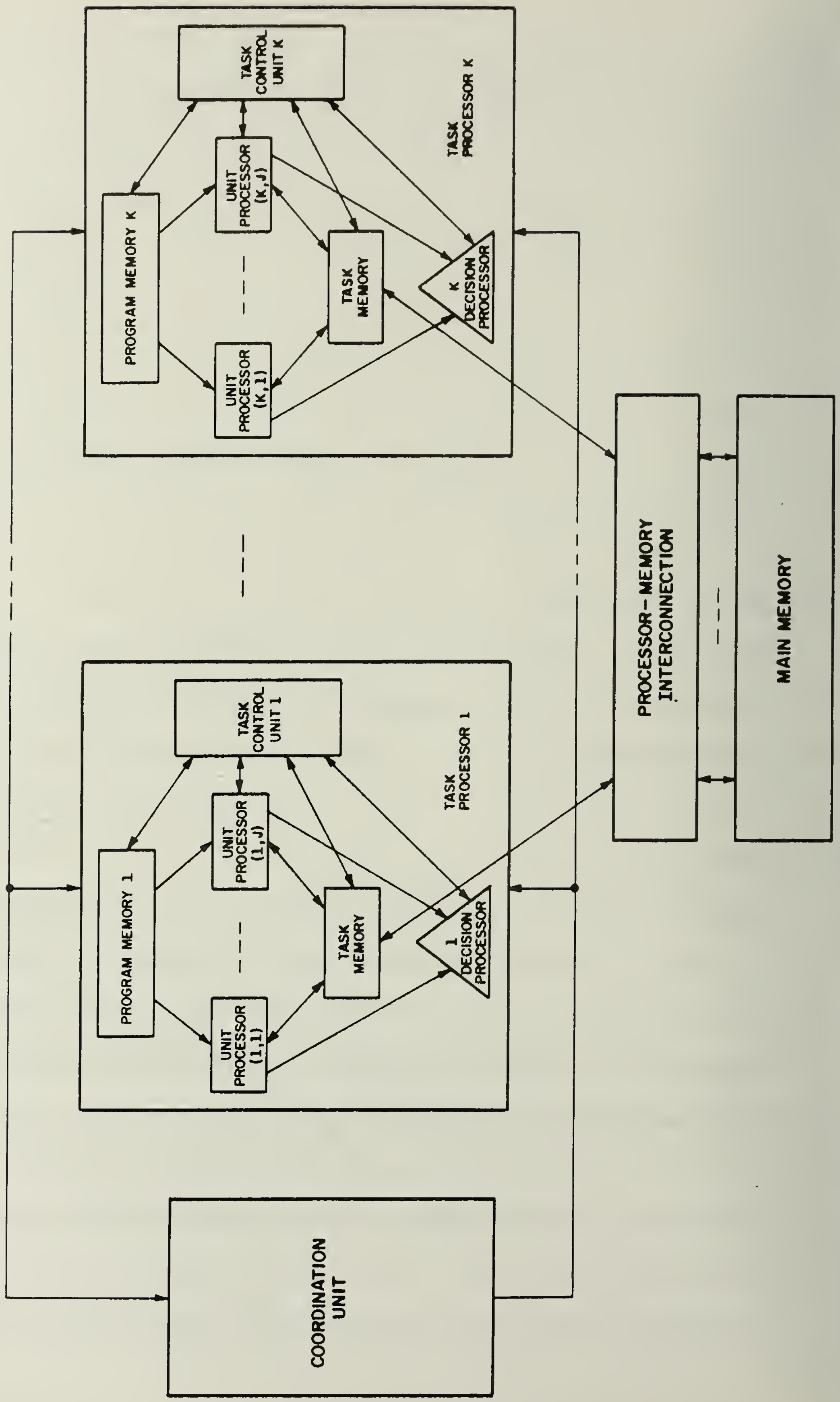


Figure 4.1. Machine Organization

the user's GPSS deck is ready to execute. The machine of this chapter requires the processing code defined in section 3.3.3. The GPSS deck must go through phases which approximate the GPSS/360 assembly and input phases. These phases include the use of the partitioning algorithm and system variable table lookup to assign the processing code to each block, the small amount of compilation for GPSS VARIABLE definition cards, and the input of block operands to distinguish the data and execution options for each occurrence of each block type. Processing of the source decks to accomplish this is referred to as compilation in this chapter.

4.3 Task Processors

A task, the combination of a transaction executing a particular block, has been identified as a part of a GPSS program which is executed in its entirety. Let a task processor be a processor capable of executing any GPSS task. To make use of the parallelism within a task implies that a task processor must be made up of several unit processors. To use the concurrency between tasks implies that the total machine must contain more than one task processor. In this section the design and arrangement of task processors is discussed.

4.3.1 Unit Processors

Results of analyzing several GPSS block types are given in section 3.3.2.1, Table 3.1. The number of processors required for the reported speed up ranges from two to 32. To design a task processor with the maximum number of parallel unit processors would ensure maximum speed up but result in lowered efficiencies for tasks which do not require the maximum number. To select the smallest number would keep efficiencies high but not exploit the parallelism in the block routine. A compromise value should be used. Examination of the tabulated analysis results indicates the number of unit processors per task processor should be in the range of eight to 16.

Simulation programs are interesting in the simplicity of their computational requirements. Study of the definition of GPSS block types and Fortran program equivalents shows that the instructions are very basic. Refer to Figure 3.7 for an example. Some frequently used assignment statements are addition of two operands, incrementing a variable, and simple replacement operations. Assignment statements involving more than two operands, or involving multiplication or division, are rare.

Arithmetic operations in fixed point are satisfactory. Note that time is an ever increasing integer. Queue lengths, contents of storages, logic switch states, parameter values, constants, etc. are all integers. In fact, all standard numerical attributes are integers when used as operands. There are a few cases where non-integer values occur. These cases use fractional values with up to six decimal digits. A floating point variable is specified in which calculations are performed in floating point but the result is converted to an integer. Since results larger than the fixed point range cannot be used it is a minor concession to restrict the design to fixed point.

A frequently occurring instruction is the conditional jump. It is used extensively to check for error conditions as well as make decisions on the model status. The frequent use of conditional jumps, with reasonably few assignment statements interspersed, was the inspiration for the decision processor of Chapter 2. Discussion of the relation between task and decision processors is covered in section 4.3.2 below.

Several conclusions on unit processor design, based on knowledge of variable and instruction types, can be reached. An unsophisticated processor with a small instruction set is satisfactory. The processor needs the ability to add, shift, fetch and store operands, decode the instruction set, etc. To use the decision processor it is necessary to be able to read the sign of the accumulator and to know if the accumulator magnitude is greater than zero. These two quantities can be used to determine the boolean value of a relational expression as explained in section 2.3.2.1.

Task processors are independent of each other in the execution of tasks. They operate asynchronously under control of the coordination unit. Unit processors that make up a task processor are all working on the same task and thus are not independent. They are synchronized in the execution of instructions by one control unit per task processor.

The task control unit has several functions. It is used for starting the unit processors and transferring instructions from the program memory to the unit processors. It communicates with the coordination unit when it is available and when it has completed a task. It controls the decision processor and access to the main memory.

Unit processors are not constrained to the alternatives of executing the same instruction or being idle. Each unit processor has instruction decoding logic. Each block type is formulated for parallel execution when compiled. At every program step appropriate instructions are transferred from the program memory to all unit processors for decoding and execution.

4.3.2 Decision Processor Use

In this section the decision processor of Chapter 2 is merged into the task processor design. From the definition of GPSS block types it was known that a significant amount of the processing was decision making. This was confirmed by the Fortran analyzer, used to study Fortran equivalents of GPSS blocks. Chapter 2 can be referred to for details on IF trees and the decision processor.

Results of block analyses show that IF trees of considerable length exist in many blocks but the number of trees per block is one or two. One decision processor of six to eight level capability associated with each task processor should be sufficient.

The decision processor accepts boolean values corresponding to arguments of nodes in the tree. The boolean values come from the unit processors with the requirement that values for a specific node in the IF tree load into the corresponding bit in the decision processor input node register. The number of nodes in the decision processor is greater than the number of unit processors. Thus several instruction cycles may be required to determine all boolean node values. It is proposed that each unit processor include a node output register to store the node values until all have been determined. The register should be of a length such that the sum of the bits for all unit processors in a task processor is at least equal to the number of decision processor nodes. One transfer from the node output registers to the decision processor is then sufficient.

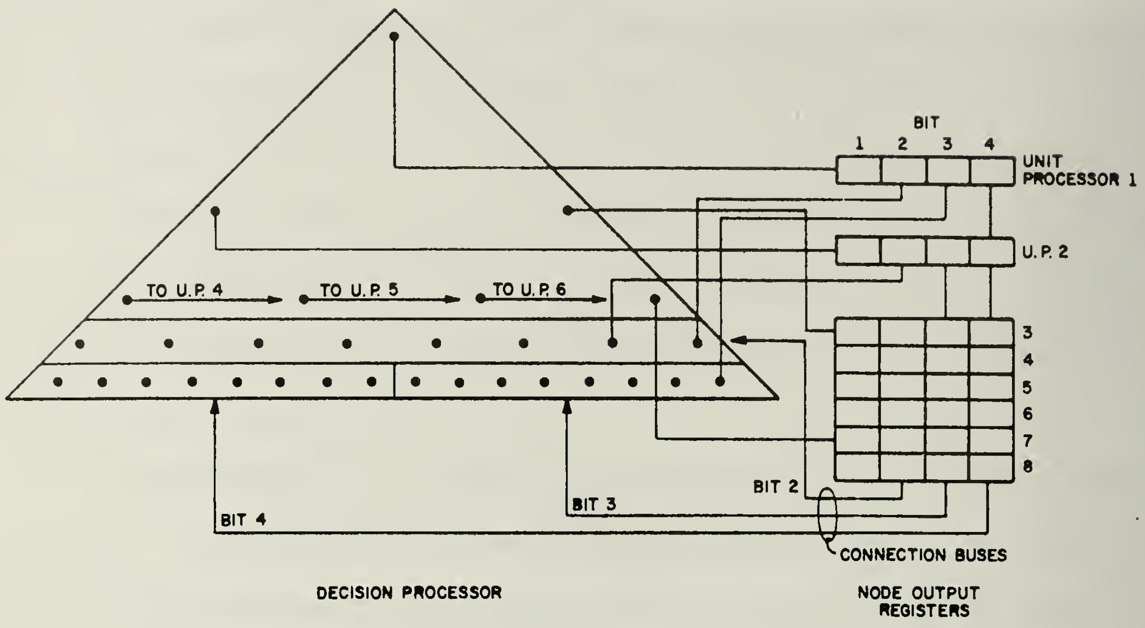
Some manner of connection must exist between the node output registers and the decision processor. Maximum flexibility is achieved with an expensive crossbar switch type of connection. A more reasonable solution is to provide simply two alternative connection schemes based on the shape of the IF tree. Each bit in the node output register of each unit processor connects to at most two bits in the decision processor input node register.

Before presenting the connections it is noted that the shape of trees can be changed somewhat without changing their logical structure. This can be done by complementing the logical argument at a node and interchanging the true and false paths out of the node. For example, IF $(A > B)$ THEN X; ELSE Y; can be rewritten equivalently as IF $(A \leq B)$ THEN Y; ELSE X;. Using this technique enables the shape of an IF tree to be biased such that the longest path out of any node is always on the right when pictured as in Chapter 2.

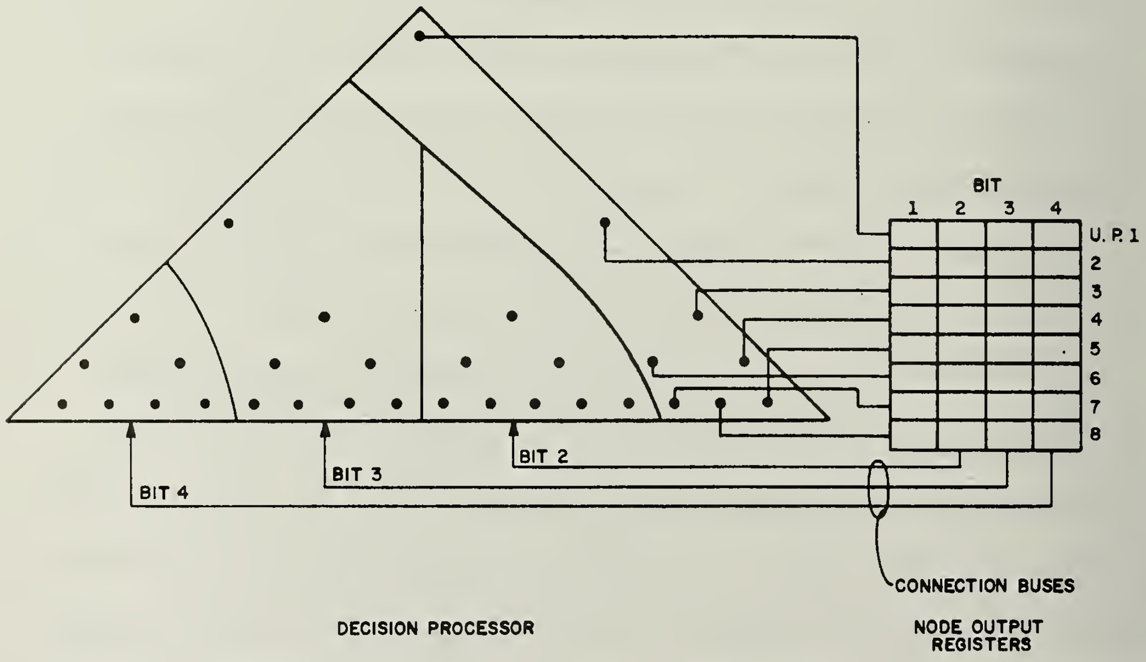
Now the two connection schemes will be explained. For trees that can be thought of as balanced, or triangular in shape, a connection is suggested in which the node values produced by the unit processors fill levels of the decision processor tree structure. For unbalanced trees, those in which a few paths are long and many are short, the connection is designed to fill the tree in a right to left sweep.

An example of the two connection schemes, "level" and "biased", is given in Figure 4.2 under the assumption that eight unit processors are to be connected to a five level, 31 node decision processor. To provide for the use of all nodes each unit processor must calculate a maximum of four node values. The node output register is thus four bits long.

The connection problem is really one of distributing the calculation of node values over unit processors in an attempt to minimize the number of values serially determined by any one processor. Clearly, for the example shown, the level scheme involves only one node calculation per unit processor for a full three level tree but three node calculations in processor one for a five level linear tree. Use of the biased connection requires only one node calculation in five of the unit processors for the same linear



(a). Level Connection Scheme



(b). Biased Connection Scheme

Figure 4.2. Connections Between Decision and Unit Processors

tree. The connection scheme for a given tree is selected at compile time. The criterion for selection is that the number of node values which must be calculated by any one unit processor should be minimized.

The original philosophy of the decision processor was to make better use of existing arithmetic processors when a program had many conditional jumps. It was assumed that any multiprocessor application would require a reasonably large number of processors. An interesting result of the Fortran block analysis was that, excluding decision trees, the required number of processors was small. For most programs the number was two to four. The number of unit processors in this machine is in the range of eight to 16 because of the IF trees. It is reasonably accurate to say that the block programs consist primarily of IF trees.

4.3.3 Task Processor Configuration

Each task processor is capable of processing any block type. The parallelism within a block is exploited by providing parallel unit processors and a decision processor within the task processor. The number of tasks that can be in execution simultaneously is dependent on the concurrency between blocks and transactions. The number is the sum of the concurrently executable blocks per transaction, taken over the number of concurrently moving transactions.

Section 3.3.3 showed that the number of blocks per transaction was slightly over two for six significantly different GPSS programs. The number of transactions which can be moved concurrently cannot be measured by study of the program. It is a run time function of the program being executed on this parallel machine. A simulator of the machine was written, primarly in GPSS, to measure the transaction concurrency of actual GPSS programs. Details of the simulation system are given in the Appendix. The results are given here.

Programs were tested in the simulator under the condition that executing tasks uses an amount of time determined by the analysis of Fortran equivalents for GPSS blocks. In the first test the coordination unit, which distributes tasks to processors, was assumed to take zero time to operate. This measures the best performance that can be expected. The results are given in Table 4.1 as the rows marked (1).

Column headings for Table 4.1 have the following meanings:

(1) The program name corresponds to the names used in Table 3.5.

(2) "Total transactions" is the number of transactions that were active at some time in the simulation. For the thesis example program it can be seen in Figure 3.5 that 24 transactions existed although only 10 went through the entire program.

(3) "Blocks executed" is the number of blocks for which execution was simulated.

(4) Beginning with this column there are multiple entries for each test program. The number of task processors is a design parameter for the machine. Simulations were run with four, eight, and 16 processors to observe the effects on the entries in columns (5) and (6).

(5) "Concurrent transactions" is the number of transactions which were being processed simultaneously. Entries were generated in the simulator by sampling transactions being processed at a time interval equal to the shortest block execution time.

(6) This column lists the simulated execution time.

A second test was run in which operation of the coordination unit

Program Name	Total Transactions	Blocks Executed	Task Processors	Concurrent Transactions	Execution Time, μs
Rail Fleet	56	613			
(1) zero time C.U.			4	2.8	2513.6
			8	4.5	1505.3
			16	6.2	1041.2
(2) real time C.U.			4	2.6	2649.6
			8	3.9	1659.7
			16	5.2	1216.0
Elevator	12	347			
(1) zero time C.U.			8	3.1	881.0
			16	3.2	820.5
(2) real time C.U.			8	2.7	1020.9
			16	2.8	956.0
Thesis Example	24	164			
(1) zero time C.U.			8	2.2	606.3
			16	2.6	526.2
(2) real time C.U.			8	2.0	667.6
			16	2.2	596.6

Table 4.1. Simulation Results: Concurrent Transactions

took realistic amounts of time. The purpose of this test was to determine the capability of the coordination unit to keep task processors busy. It also shows the degradation to the concurrency in the program caused by a finite time span between completion of one partition and the start of the next. The entries for this test are marked (2) in Table 4.1.

Analysis of GPSS programs in Chapter 3 gave very similar results on average partition lengths for six different programs. Results presented in Table 4.1, for three of the programs analyzed in Chapter 3, give quite different results. The number of transactions which can move simultaneously is obviously not strongly related to static program analysis results.

The Rail Fleet and Elevator programs have approximately the same number of source blocks, including TRANSFERs. The maximum and average partition lengths are close. The difference, in actually running the program, is the total number of transactions. Clearly the program with more transactions can have more concurrency between them. The Thesis Example program has the intermediate value on the total number of transactions and the lowest value of concurrent transactions. This program however is only one-fifth as long, in terms of source blocks, as the other two.

For programs which appear comparable, such as Rail Fleet and Elevator, a given factor of difference in the total number of transactions does not yield the same factor in concurrent transactions. One important reason for this is that the scheme used to allow concurrent transactions is more restrictive than the language requires. As currently designed into the processing code assignment, and implemented by the coordination unit to be described, any block which reads a system variable must be executed by a transaction which is at the minimum simulated time in the model.

The actual requirement is that a system variable must not be read by a transaction if its value can be changed by a transaction at a lower simulated time. A small improvement can be made without changing any of the proposed hardware by using a more sophisticated program for assigning the S bit of the processing code. The required change would be to note those system variables whose values are never changed in the program. These system variables should not cause a block to be assigned S = 0.

To break loose the transaction movement bottleneck a major change is needed. A transaction which uses a system variable needs to be delayed only until it is known that no transaction at a lower simulated time can change that variable. The run time information on the model status required to do this is much greater than what is needed for the machine presented here. This implies much greater complexity in the coordination unit.

The machine organization conclusion drawn from Table 4.1 is that eight task processors are reasonable. Limiting the design to four task processors imposes a physical limit of four on transaction concurrency and increases execution time significantly. There is improvement in execution time going from eight to 16 but it is not as great, even in absolute terms, as the improvement in going from four to eight task processors.

4.3.4 Hardware Design Considerations

Design objectives for this machine are to use currently available, familiar, inexpensive parts and balance the speeds and bandwidths of all system components. As a starting point it is assumed that the task processor is designed with TTL gates having propagation delays near 10 nanoseconds (ns). A 5 MHz clock generates pulses at 200 ns intervals, allowing combinational logic chains up to 20 gates. Note that one cycle of an eight level decision processor takes just one clock.

If instructions take an average of two or three clocks the instruction execution time averages out near 500 ns. Table 3.1 gives a count of instruction steps for the execution of many GPSS block types on a multiprocessor capable of executing p operations simultaneously. Using the 500 ns average instruction execution time, average block execution times can be stated. These times are used in the simulator described in the Appendix.

Now consider the amount of hardware required for the task processor configuration, excluding memories and their interconnections. In summary there are eight to 16 unit processors, one decision processor of six levels, and one task processor control unit. The unit processor is a simple device with on the order of 1000 gates and flip flops. From Table 2.2 the six level decision processor, including total sector control, has about 500 logic circuits. Estimate the task processor control unit at 3500 gates. A task processor with eight unit processors has on the order of 12,000 gates. For 16 unit processors the number is 20,000. A configuration of eight task processors requires approximately 160,000 gates.

4.4 Coordination Unit

This unit is concerned with the selection of tasks for execution in the task processors. Selection is based on the simulation model status, the transaction time of all transactions, and the SP processing code assigned to each block in the simulation program. A transaction available for movement is selected, then tasks are formed from the sequence of blocks the transaction will move through.

In GPSS/360 the only transactions selected for movement are those on the current events chain. The equivalent of this chain is the set of run time transactions, those which are farthest behind in simulated time. The selection of transactions is still oriented toward the min time set but will, as explained in section 3.3.2.3, select transactions which are not at the min time. When all transactions in the min time set are in the process of being moved, the next transaction to move is chosen from the set nearest to the min time. This selection method reduces the time spread on transactions being processed and minimizes the conflicts caused by overlapped times.

Recall that blocks with S = 0 can be processed only by min time transactions. By always working on transactions which are min time, or near to it, the min time (which represents the upper limit of completed simulation time) advances as rapidly as possible. The result is that the real time spent by transactions at blocks with S = 0, waiting for the model to "catch up" to them, is reduced. Thus selection of the transaction to move is based on the transaction time with smaller values being selected.

Following selection of a transaction, a task is set up for each block in the partition the transaction is in. Part of the transaction status information is a word which identifies the next block it is to process. Sequential blocks are in the same partition until the code bit P = 0. The tasks formed fall into the categories of being available for processing immediately or being available when the transaction becomes a min time transaction. Two queues, called "process" and "delay" respectively, are maintained for the two categories.

Task formation and routing to one of the two queues is controlled by the processing code bits, SP, assigned to each block in the program. The interpretation of the code is given now. The significance of S = 0 is that the block uses a system variable and processing must be delayed until the transaction time is the minimum of all transaction times. The significance of P = 0 is that the block is the last in a partition of simultaneously executable blocks. The transaction cannot continue moving until the current partition has been processed. The action taken by the coordination unit when examining the block code is listed in Table 4.2.

Code S	Code P	Coordination Unit Action
1	1	Add task to process queue. Examine next block.
1	0	Add task to process queue. Select next transaction.
0	1	Add task to delay queue. Examine next block.
0	0	Add task to delay queue. Select next transaction.

Table 4.2. Processing Code Interpretation

The actual implementation has one modification. All tasks in a partition following the first one that is added to the delay queue are also added to the delay queue. The reason is that blocks which use system variables are assigned S = 1 if a prior block has been assigned S = 0 and there can be no change in transaction time between the two blocks.

In summary, the coordination unit contains logic to select the transaction with the minimum value of simulated time, to examine the code for the block it is to process next, and to place the transaction-block pair, a task, in one of two queues. The process queue receives tasks that can be executed as soon as a processor is available. The delay queue receives tasks which must wait for the min time of the model to reach their scheduled event time. The coordination unit releases tasks from the queues for distribution to the processors. Figure 4.3 is a diagram of the unit. Discussion of the components is covered in following sections starting with the selection of a transaction to move.

4.4.1 Transaction Selection

Transactions in this machine can be in several states, some of which make a transaction ineligible for movement. The logic of this part keeps a record of the state of all transactions and allows selection of movable ones according to their simulated time ordering.

When a transaction is selected for movement, all blocks in the current partition are set up as tasks for processing. The transaction is not eligible to move again until all tasks in the partition are completed. A transaction in this state is considered "selected". A transaction may reach a blocking condition, defined in section 3.3.1. Such a transaction cannot move, but note that its simulated time is implicitly updated to the time when the blocking condition is removed. A similar situation exists for transactions which are put on user chains by means of the LINK block. They cannot be moved until they are removed from the chain by an UNLINK block. Transactions which are blocked or are on user chains are considered in the same state with respect to selection for movement. A transaction in this state is both "selected" and "blocked". A third state is the transaction which has entered the delay queue. The minimum simulated time in the model must be known so that transactions in this "delayed" state can be removed from the queue.

All of the above transactions are in the state of having been selected for movement. The final state consists of all unselected transactions. It is from these that one must be selected for movement.

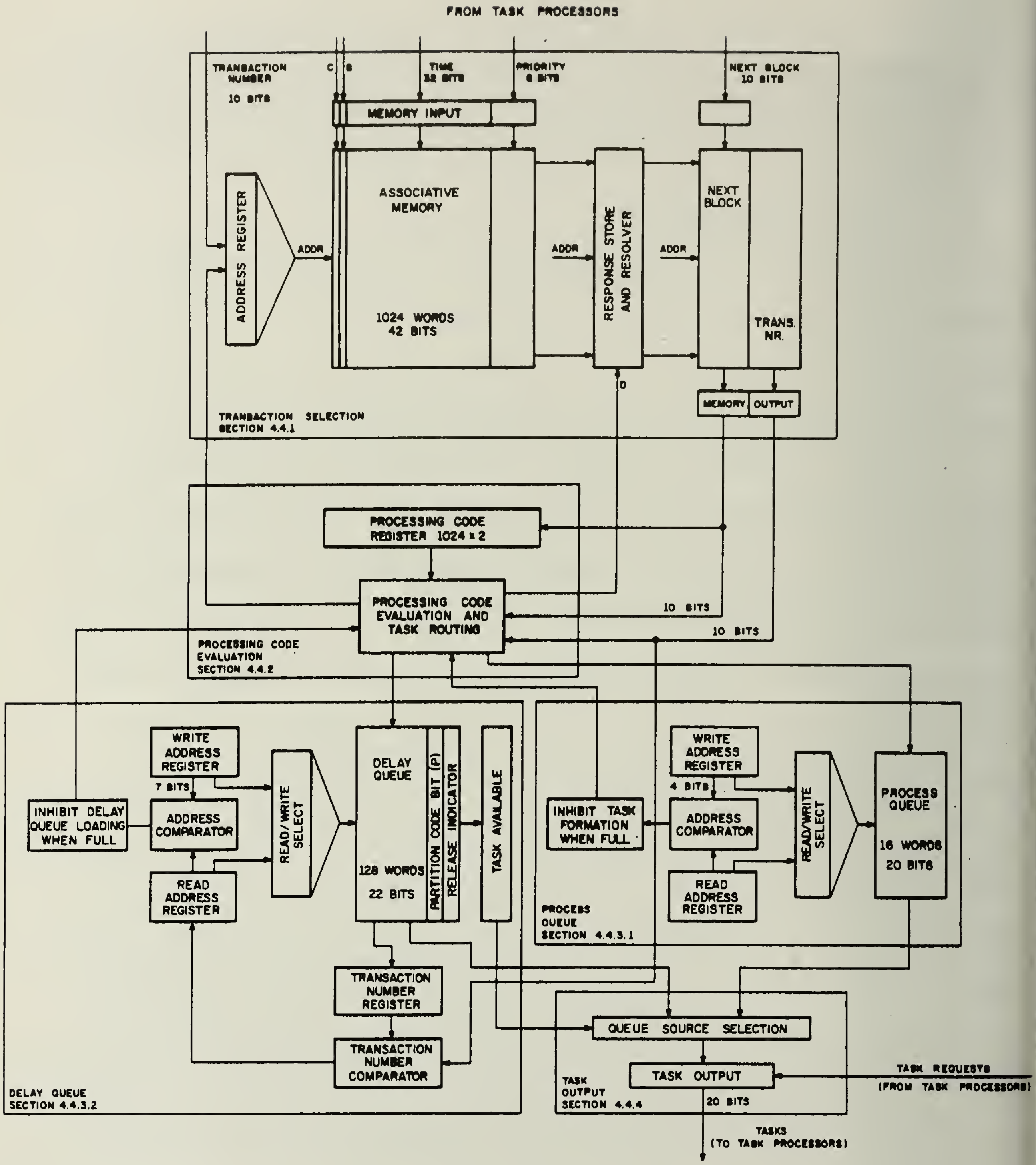


Figure 4.3. Coordination Unit

The logic in this part is primarily a moderate size memory on which a minimum value search can be performed associatively. One word per transaction is required. The number of words in the memory is the number of transactions defined as the maximum for the particular machine. The normal GPSS allocation is 600 transactions for the 128K memory size and 1200 for the 256K or higher. Thus a 1024 word memory would be adequate for quite large simulations.

Each word must contain fields for the transaction time, priority, and status indicators. The time and priority fields in GPSS use 32 and eight bits respectively. These fields, plus two bits to indicate "selected," C, and "blocked," B, transactions give a word length of 42 bits. The format for each word is given in Figure 4.4.

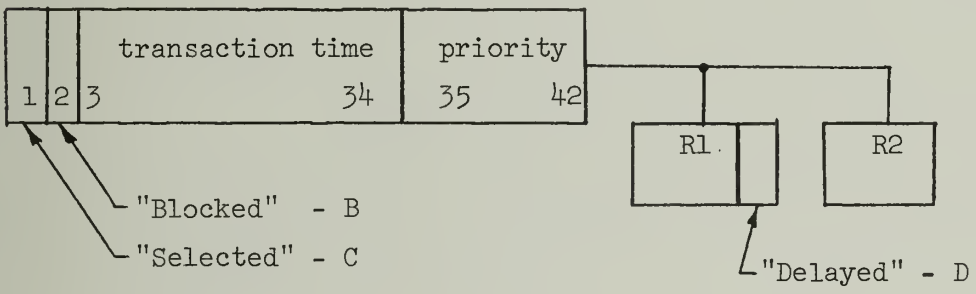


Figure 4.4. Word Format of Transaction Status Memory

Priority is an extension of the time field on the least significant end. If a low value in the priority field is defined to represent a high priority, the highest priority transaction can be found by continuing the minimum value search through the priority field.

The C and B control bits are at the most significant end of the time field. C is set to one when a transaction is selected for movement. It is reset when processing of its partition is complete and it becomes eligible for selection again. B is set to one when a transaction is blocked or put on a user chain. It is reset when the blocking condition is removed or the transaction is removed from the user chain. If either bit is one the transaction will not be a responder to a minimum value search unless there are no unselected transactions.

The "delayed," D, bit is not part of the minimum value search. The D bit is set when a transaction is routed to the delay queue. It is reset when the transaction is removed from the queue.

Each word has two response indicators, enabled on slightly different search bits. The first, R1, is enabled for bit B and the time and priority fields. Responders indicated by R1 are transactions in the min time set, whether previously selected or not. The second response indicator, R2, is enabled for bit C in addition to the bits for R1. R2 responders are non-selected transactions with the smallest value of time.

Any R1 responders which are also in the delay queue are now eligible for execution. Bit D is ANDed with R1 to eliminate responders that are not in the delay queue. The D bit for all remaining responders is reset since the transactions will be released from the queue.

After releasing transactions from the delay queue, responders in R2 are selected for movement. The action upon selection is covered in the next section. When all R2 responders have been selected the interrogation of the memory begins again.

Clearly it is necessary to update the memory when the status of any control bit, or value in the time or priority fields, changes.

4.4.2 Processing Code Evaluation

Each transaction has a word identifying the block into which it will move next. The identifier is the number of the block in the program. In the coordination unit the next block number is used as a pointer to the needed processing code register entry.

When a transaction is selected for movement, the processing code for the next block it is to execute is examined and interpreted according to Table 4.2. Sequential block codes are examined until P = 0, indicating the end of the partition and need for the next transaction.

For each block in the partition a task is set up and routed to either the process or delay queue, described below. If a task is routed to the delay queue, the D bit for that transaction is set to indicate the transaction is being delayed. A task is fully defined by the transaction and next block numbers. Since 1024 is the maximum number of both transactions and blocks, 10 bit fields are required for each. In the task processors the next block number is used as the address of a word which identifies the block type and the particular occurrence of this type.

4.4.3 Task Queues

4.4.3.1 Process Queue

Tasks in this queue are available for processing at the request of task processors. Each entry is a transaction number and the number of the block it is to execute. At 10 bits for each of these numbers, an entry is 20 bits.

This queue provides a backlog of tasks for the processors. When it becomes full the formation of tasks can be halted. The queue only needs to be long enough to assure that it cannot be emptied before task formation can resume. For a machine with eight task processors a queue of length 16 is easily long enough.

Design of the queue is circular with a first-in first-out discipline. Separate read and write address registers point to the oldest entry and the first empty location. The registers are incremented following every read or write. An address comparator detects a full queue and inhibits the formation of any more tasks.

4.4.3.2 <u>Delay Queue</u>

Tasks which must be processed in a simulated time order, because of their use of system variables, are routed to this queue. Processing is delayed until the transaction becomes a min time transaction. The release of tasks is controlled by the transaction selection logic which interrogates the time value of all transactions to identify those in the min time set.

Recall that all blocks in a partition following the first block with S = 0 were also routed to the delay queue. Thus when a min time transaction is found to be in the delay queue, the task that caused transaction movement to be delayed is released, as well as subsequent tasks in the same partition. The processing code P bit is used to indicate these tasks.

Tasks in the delay queue can be in the state of having been released but not yet removed from the queue for processing. An indicator is needed to distinguish these tasks from those not yet released. Each entry in this queue needs, therefore, the 20 bits that identify a task, a bit for the P portion of the processing code and a bit to indicate released tasks. Actual setting of the release bit is based on a comparison of the number of the transaction to be released and transaction numbers in the queue.

This queue holds tasks which will become available for processing when the model reaches the condition that no transaction has a simulated time less than that of the transaction being delayed. The queue length in this case must provide for potentially many tasks. It is suggested that this queue be of length 128. When it becomes full, further loading must be inhibited. The design is similar to the process queue except the removal of entries is based on matching transaction numbers rather than length of time in the queue.

Movement of tasks from the queues to the task processors is controlled by the task output logic described next.

4.4.4 Task Output

The two sources of tasks for output to the task processors are the process and delay queues. The overall processing algorithm is to move transactions which have the smallest time values. Transactions in both queues have time values which range upwards from the minimum completed time. Tasks in the delay queue which are marked as released belong to the min time transaction set by definition. These tasks have priority over tasks from the process queue for output.

If any release indicator bit in the delay queue is set, that queue is the source of tasks for output. When all released tasks have been output for processing, tasks are taken sequentially from the process queue.

The task output logic communicates with the task processors. It accepts requests for tasks. It identifies, for the requesting processor, the transaction and next block number that make up the task.

4.4.5 Coordination Unit Hardware

This unit must be able to form and dispatch tasks at a rate to match the task processor consumption. While the number of task processors is unlimited in theory, this unit will place a limit on the number it can support. The design and hardware suggested here is for the eight task processor configuration proposed in section 4.3.3. Suggestions will be given for increasing the number.

A pipeline effect exists in the selection of transactions and formation of tasks. Transactions which have been moved through the blocks in a partition enter at the top of the coordination unit with time and next block data. They filter through the unit to exit as tasks for further block movement. The memory can be updated while responders from the previous search are being resolved and examined by the code evaluation logic. Note that one interrogation may yield more than one responder and each responder will yield an average of two tasks. Loading of the two queues can be interleaved with unloading by the task output logic.

Average task execution time, as determined from Table 3.1 with the assumption of 500 nanoseconds (ns) per instruction, is 10 microseconds. For eight task processors, with full utilization, the task output rate should therefore average one task per 1.25 microseconds. With ordinary TTL logic having typical propagation delays of 10 ns the task output logic will have no trouble satisfying that rate.

In the worst case for the process queue, it must be both loaded and unloaded at that rate. Since the two actions cannot be concurrent, the rate for either must be near 600 ns. Clearly there will be adequate time to decode a four bit address and either read or write a 20 bit word.

The delay queue is more complicated. Loading is similar to the process queue but unloading requires transaction number comparisons. When the transaction selection logic has an R1 responder, that transaction number is known to be in the delay queue. Entries in the queue are not required to be in simulated time order but will tend to be so due to the selection algorithm. The comparison begins with the oldest entry and moves sequentially through the queue. When the matching transaction is located the task is released. All other tasks for this transaction, identified as all subsequent tasks in the queue until the partition code bit, P, is zero, have the release indicator set. They are thereby marked as executable.

The transaction location sequence of reading, followed by a comparison and either increment of the read address register or release of a task, is a two clock sequence. If the process queue is empty and all tasks must come from the delay queue, the release rate must be the same as the task output logic, 1.25 microseconds per task. At this point a 100 ns clock rate is specified for the coordination unit. At two clocks to load a task, 200 ns are used. This leaves 1000 ns to locate and release a task. Thus five transaction number comparisons can be performed in the allotted time. This is thought to be quite adequate.

The conclusion, with respect to the delay queue, is that it is capable of being loaded and unloaded in the required time even when it is the only source of tasks.

Feeding the two queues is the processing code evaluation logic. The output requirement here is again an average of 1.25 microseconds. There is clearly no time problem if selected transactions can arrive at a fast enough rate. Since there is an average of two tasks per partition, the arrival rate must be one transaction every 2.5 microseconds from the transaction selection logic.

A bit serial interrogation at the rate of 100 ns per bit requires 4.2 microseconds to cover the 42 bit associative memory. Thus transaction selection appears to limit the operating speed of the coordination unit. Comments here will show that the situation is not desperate. First note that it is possible to have more than one transaction respond to a search. If there is an average of two responders, the output rate is satisfactory.

In general it is not necessary to interrogate all bits to determine the minimum value. Interrogation can cease when the minimum is found thereby reducing the search time. For a minimum value search all words are initially considered responders and the interrogation begins with the most significant bit. It is seldom that the discrimination which produces the final responders occurs in the most significant bits. For this reason it is possible to group the interrogation of these bits. Suppose for example the 16 most significant were interrogated in parallel. If responders still exist, serial interrogation continues with a savings of 15 interrogation times. If no responders exist, it is necessary to re-initiallize the response stores and begin a bit serial interrogation at the most significant bit. The time penalty has been one interrogation plus initiallization.

Another scheme which reduces the number of interrogations and increases the number of responders is based on the fact that R2 responders need not be true minimums. By always stopping the interrogation a small number of bits short of the least significant bit, a cluster of transactions near the minimum is selected. It was noted earlier that just two responders eliminated the time problem.

Finally, note that the original calculation was based on the maximum task output rate.

The conclusion here is that the proposed coordination unit hardware, implemented with 10 ns logic, will perform with sufficient speed to support eight task processors. Suggestions have been given on ways to increase the speed in the area which appears to be most limiting. The estimated number of logic circuits in the unit is 10,000 plus the memory chips.

4.5 Memories

Data and program memories are separated in this machine and will be discussed separately.

4.5.1 Data Memory

Processors are organized as a cluster of unit processors to form a task processor, then a cluster of task processors. The data required by a task processor is the data associated with the transaction and with the block for the task. The memory organization given here uses a small memory at each task processor for the data required for that task and a main memory to which each task processor has access. The main memory provides the only communication between task processors.

Execution of a task involves a transfer of all data related to the task from the main memory to the task memory, followed by restoration of variables changed in the execution. The maximum amount of transaction data in GPSS/360 occurs when 100 full word transaction parameters are used. In this case the total transaction data is 436 bytes. Blocks use a basic allocation of 12 bytes plus four bytes per operand. There is a maximum of seven operands per block with the average number near three. At four bytes per operand the average allocation is 24 bytes. Transaction and block data, totalling 460 bytes for maximum transaction parameter allocation and typical blocks, must be transferred to each task processor for each task.

Each task may also use system variables which are accessed directly from the main memory. The number of such variables per block is small. To design for 10 variables, or 40 bytes, would be liberal. Total task data is then 500 bytes.

Given this background, the data memory sizes, bandwidths, and interconnections are discussed in this section.

4.5.1.1 Main Memory

The starting assumption for calculations on the main memory is that the time required to transfer task data from main to task memory is equal to the compute time. The time to restore changed variables is negligible.

Average task execution time is 10 microseconds (μs). The bandwidth per task, for maximum size transactions, is

$$\text{B.W.}_{\text{main-task}} = \frac{500 \text{ bytes} \left(8 \ \frac{\text{bits}}{\text{byte}}\right)}{10 \times 10^{-6} \text{ seconds}} = 400 \times 10^{6} \text{ bits/second.}$$

The total main memory bandwidth, for eight task processors, is

$$\text{B.W.}_{\text{main}} = 8 \times 400 \times 10^{6} \text{ b/s} = 3200 \times 10^{6} \text{ b/s.}$$

An ordinary core memory can supply a 64 bit word in 0.5 μs. To achieve the required maximum bandwidth the number of memories is

$$M_{\max} = \frac{3200 \times 10^{6} \ \frac{\text{b}}{\text{s}}}{128 \times 10^{6} \ \frac{\text{b}}{\text{s}}\Big/\text{memory}} = 25 \text{ memories.}$$

Design of the machine with 32 main memory units more than satisfies the maximum bandwidth requirements. A 16 memory design provides a bandwidth of

$$B.W._{16} = 16 \times 128 \times 10^6 \text{ b/s} \sim 2000 \times 10^6 \text{ b/s}.$$

Each task processor would receive one-eighth of this, or 250×10^6 b/s. This leads to a reduction of 200 bytes in the task bandwidth, or 50 full word parameters. Thus 16 memories can supply 50 fullword or 100 halfword parameters.

The choice between 16 and 32 memories is a design option. The lower number is satisfactory for a high percentage of actual simulations.

Total variable memory size is somewhat flexible. The largest GPSS/360 option begins at 256K bytes and extends upward to the core limits. Suppose 400K bytes were chosen for a large system using 16 memories. Each memory is 25K bytes, or less than 4K 64 bit words.

Main memory is therefore 16 modules of 4K word, 64 bit memories, with a 0.5 microsecond cycle time. Optionally, 32 slower modules could be used.

The switch to connect this memory to the task processors is a fanout tree. Such a tree has on the order of three gates per bit per destination. The number of bits in the path is the product of the number of memories and the bits per memory, or $16 \times 64 \sim 1000$. The eight task processors are the destinations. The gates in the switch thus total $3 \times 1000 \times 8 = 24K$.

4.5.1.2 Task Memory

Each task memory must be capable of storing the data for the current task plus the data being set up for the next task. That is, it must have storage for twice the maximum task data requirement. The maximum amount of task data is 436 bytes for the transaction and 40 bytes for the block for a total near 500 bytes. Let the memory be 1K bytes.

The bandwidth of data moving between this memory and the main memory was previously calculated as $B.W._{\text{main-task}} = 400 \times 10^6$ bits per second. The task memory must simultaneously provide operands for the task processor. The number of memory accesses by a block program with the 10μs average execution time is near 50. If these are all 32 bit words, the memory to processor bandwidth is

$$B.W._{\text{task-proc}} = \frac{50 \text{ words } \left(32 \frac{\text{bits}}{\text{word}}\right)}{10 \times 10^{-6} \text{ seconds}} = 160 \times 10^6 \text{ b/s}.$$

This bandwidth is satisfactory only if there are no conflicts on accessing memory. Conflict free access would require one memory capable of supplying the total bandwidth going between the task memory and both the main memory and the task processors. That is 560×10^6 b/s. For 32 bit words, the cycle time would have to be

$$t_{\text{cycle},1} = \frac{32 \text{ bits}}{560 \times 10^6 \text{ b/s}} \sim 60 \times 10^{-9} \text{ seconds}.$$

A 1K byte memory with a cycle time of 60 ns per 32 bits does not meet the design objective of inexpensive parts. To lower the cycle time an array of eight memories is suggested. The memories should have 32 bit word lengths. A total of 250 words is needed so each memory has only 32 words. The connection to the task processors is then a crossbar switch; a not unreasonable connection for the small numbers involved. For example, connecting eight processors and memories with a 32 bit path takes 8 x 8 x 32 ~ 2K gates. If the design uses 16 unit processors the connection takes 4K gates.

Memory conflicts are possible with this modular scheme. Recall from section 4.2 that compilation of routines can finely tune the code produced so as to minimize conflicts. Assume however that conflicts do occur one-fourth of the time, increasing the required bandwidth between the memories and processors to 200×10^6 b/s. Total task memory bandwidth is then 600×10^6 b/s. The cycle time is

$$t_{cycle,8} = \frac{32 \text{ bits } (8 \text{ memories})}{600 \times 10^6 \text{ b/s}} \sim 400 \times 10^{-9} \text{ seconds.}$$

A 400 ns cycle is within the capabilities of current memory devices.

The task memory, in summary, is an array of eight memories, each with 32 words and 32 bits. The cycle time of each is 400 ns. The memory is connected to the unit processors by a crossbar switch.

4.5.2 Program Memory

Section 4.2 mentioned that a GPSS program is a series of calls for execution of the routines that correspond to blocks. The routines are compiled one time and remain valid until the language definition is changed. Since this executable code does not change it can be stored in a read only memory.

Consider the size of the memory required to store the program for all 44 block types. From Table 3.1, the number of Fortran statements per block averages near 50. Assume a factor of four is needed to convert Fortran statements to machine instructions. Then 200 machine instructions per block, and 8800 instructions altogether, are required.

The instruction format for unit processors will now be mentioned. The instruction set is small and will be fixed at 16. Four bits are needed for the operation code. Unit processors communicate with the task memory, a 256 word memory requiring eight bits to address. A single address instruction format therefore uses 12 bits. Assume four additional bits can be used, as for indirect addressing, giving a total of 16 bits per instruction. A long instruction is defined as one suitable to drive all the unit processors within a task processor. It is 128 bits for the eight unit processor design, or 256 bits for 16 unit processors.

With eight task processors, each capable of executing any task, the demand for instructions is great. Assuming a 500 ns average instruction execution time this memory must be capable of delivering a long instruction at intervals of 62 ns. Such capability fails to meet the objective of inexpensive parts. Alternatives are examined.

The program memory size is 8800 x 16 = 140800 bits. Organized as 64 bit words it is 2200 words. If instructions are distributed from a read only main program memory to local program memories at each task processor, the local memories must be capable of holding the two longest block programs. For the blocks converted to Fortran, this would be near 200 Fortran instructions or 800 machine instructions. The local memories, which must be read/write, would have a bandwidth requirement composed of program loading and instruction reading. These components are

$$\text{B.W.}_{\text{load}} = \frac{\left(200\ \frac{\text{instructions}}{\text{program}}\right)\left(16\ \frac{\text{bits}}{\text{instruction}}\right)}{10 \times 10^{-6}\ \frac{\text{seconds}}{\text{program}}} = 320 \times 10^{6}\ \text{b/s}$$

$$\text{B.W.}_{\text{read}} = \frac{128\ \frac{\text{bits}}{\text{instruction}}}{0.5 \times 10^{-6}\ \frac{\text{seconds}}{\text{instruction}}} = 256 \times 10^{6}\ \text{b/s}.$$

The combined bandwidth of 576×10^6 b/s is the same as that required for the smaller task memories.

Since the program can be stored in a read only memory it becomes attractive to consider duplicating the program at each task processor. The tradeoff is eight read/write memories capable of storing the two longest blocks against seven full program read only memories (at roughly half the bandwidth). The read only memory scheme will be chosen.

With a read only memory at each task processor the bandwidth is the $\text{B.W.}_{\text{read}}$ calculated above, 256×10^6 b/s. That is the figure for eight unit processors. If the memory is organized such that one cycle is capable of supplying one long instruction, the cycle time is the instruction execution time, which averages near 500 ns. An example of such an organization for the eight unit processor case, would be two memories with 64 bit words.

The program memory is thus 2200 words of 64 bits. It is read only with a 500 ns cycle time. The memory is duplicated at each of the eight task processors.

4.6 Machine Design Summary and Performance Estimates

Throughout this chapter estimates have been made on the hardware involved in the components. The estimates are tabulated and totaled in Table 4.3.

Estimates of program speedup and concurrency in processing have been given in various sections. These estimates will be summarized and discussed here.

The speedup in executing an individual GPSS block was determined in section 3.3.2.1. Many block types were converted to Fortran and analyzed.

Component	Option	Gates and Flip Flops	Memory words	bits/word
Unit Processor (U.P.)		1,000		
Decision Processor	six levels	500	64	12
Task Processor Control Unit		3,500		
Task Data Memory and Switch	8 U.P. 16 U.P.	2,000 4,000	256	32
Task Program Memory			2200	64 (read only
Task Processor Totals	8 U.P.	14,000	2200 140	64 (read only 64
	16 U.P.	24,000		
Coordination Unit		10,000	1024 1168	42 (associati) 20
Main Data Memory and Switch	16 modules	24,000	64K	64
Total Machine (8 Task Processors)	8 U.P.	146,000	~ 65K 17K 1024	64 64 (read only 42 (associati)
	16 U.P.	226,000		

Table 4.3. Total Machine Hardware Summary

Results are given in Tables 3.1 and 3.2. From Table 3.2 it can be seen that execution speedup ranged from one to eight and that the most frequently occurring speedup factor was four. It is reasonable to expect a speedup of four by multiprocessing individual blocks. The unit processors of section 4.3.1 provide this processing.

Further speedup can be achieved by simultaneous processing of more than one block. The number of blocks that can be processed simultaneously, related to one transaction, is covered in sections 3.3.2.2 and 3.3.3. Table 3.5 gives the results of analyzing several GPSS programs and shows, in column (5), that an average of two blocks can be processed simultaneously. This gives a speedup factor of two for each transaction which is multiplicative with the factor of four for each block.

Further speedup can be achieved by concurrently moving more than one transaction. This was studied in section 3.3.2.3 and 3.3.3. The number of concurrent transactions was measured on a simulation system described in the Appendix. The results, given in Table 4.1, vary with the program being simulated. The number, in the range of two to five, is the speedup factor due to multiple transaction processing. The configuration of multiple task processors is needed for the speedup due to processing more than one block per transaction and moving more than one transaction.

There is yet another area which contributes to the speedup of this machine over a serial machine. This area is the selection of a transaction to move and a block to execute. In a serial machine the selection is through the software overall scan algorithm shown in Figures 3.2, 3.3, and 3.4. In the machine designed here, the selection is carried out by the hardware co-ordination unit of section 4.4.

The software algorithm is used following the execution of each block. It is a fairly complicated algorithm and is estimated to take as much execution time as the actual processing of the block it selects. The hardware algorithm operates in parallel with the processing of blocks. This should realize another factor of two speedup over any serial execution machine.

Combining the speedup factors gives the total expected execution speedup of the organization presented here over a serial organization. The result covers a range due to the range of the transaction concurrency factor, f_{tc}. Excluding f_{tc} the speedup is by a factor of 16. The total speedup is by $16 \cdot f_{tc}$. The total speedup factor ranges therefore from 32 to 80. This improvement in execution speed was achieved with the use of currently available, moderately priced hardware.

5. CONCLUSION

This thesis has resulted in the design of a machine which yields a significant improvement in execution time for discrete time simulation languages similar to GPSS. A multiprocessor organization, using currently available logic and memory elements, is employed. The machine includes a device to assist in the evaluation of decision trees.

Specific results are presented at several places in the preceding chapters. The locations of the results are given for reference in the following paragraphs.

Chapter 2 was concerned with a "decision processor" for finding paths through decision trees. This device, designed for use in a general multiprocessor environment, is shown in block diagram form in Figure 2.8. Logic circuit counts are given in Table 2.2 and delays are given in Table 2.3. As an example of the values, a processor capable of evaluating trees with up to 255 nodes requires fewer than 2000 gates plus a memory on the order of 64 words by 12 bits. The same processor can evaluate up to eight tree levels in approximately one clock cycle. An example of the processor operation is given in section 2.5.1.

Simulation was first mentioned in Chapter 3. GPSS is discussed and analyzed for execution concurrency. Results of analyzing Fortran versions of 21 GPSS block types on the system described in [10] are given in Tables 3.1 and 3.2. One conclusion taken from those tables is that a factor of four speedup is possible within the blocks. Concurrency between blocks is shown to exist, providing potential for additional speedup.

In Chapter 4 a machine was designed which exploits the concurrency found in GPSS. A high level block diagram of the organization is given in Figure 4.1. Hardware and performance summaries are the subject of section 4.6. The total speedup was found to be near 50 for a sample of fairly small simulation programs.

APPENDIX

This appendix describes the software system used to generate the experimental results of the thesis. The system involves a series of programs written in PL/I, 360 Assembler Language, and GPSS. The programs do some analysis of GPSS source decks and simulate the execution of the decks on the multiprocessor designed in this thesis. Figure A.1 diagrams the system and includes references to results or discussions of the components.

A.1 GPSS Scanner

The scanner program was mentioned in section 3.3.3 on assigning the processing code to the blocks of a GPSS program. It implements Algorithm 3.1 for partitioning, and the system variable table look-up, to do the code assignment. It also generates the statistics presented in Table 3.5. Another function of this program has not been previously mentioned. It produces punched card output to modify the original test GPSS program such that a run time "trace" of the original program can be gathered. The trace is explained in the next section, which covers the program that gathers the trace data.

A.2 Trace Data Extraction: XTRAC

The goal of the complete test system is to simulate the execution of some real GPSS programs on the proposed machine. To simulate the execution it is necessary to know certain things about the actual execution. It was noted in Chapter 3 that the execution sequence for a GPSS program cannot be determined from the source code. It can only be determined by tracing the execution. Any trace should identify the block being executed and the

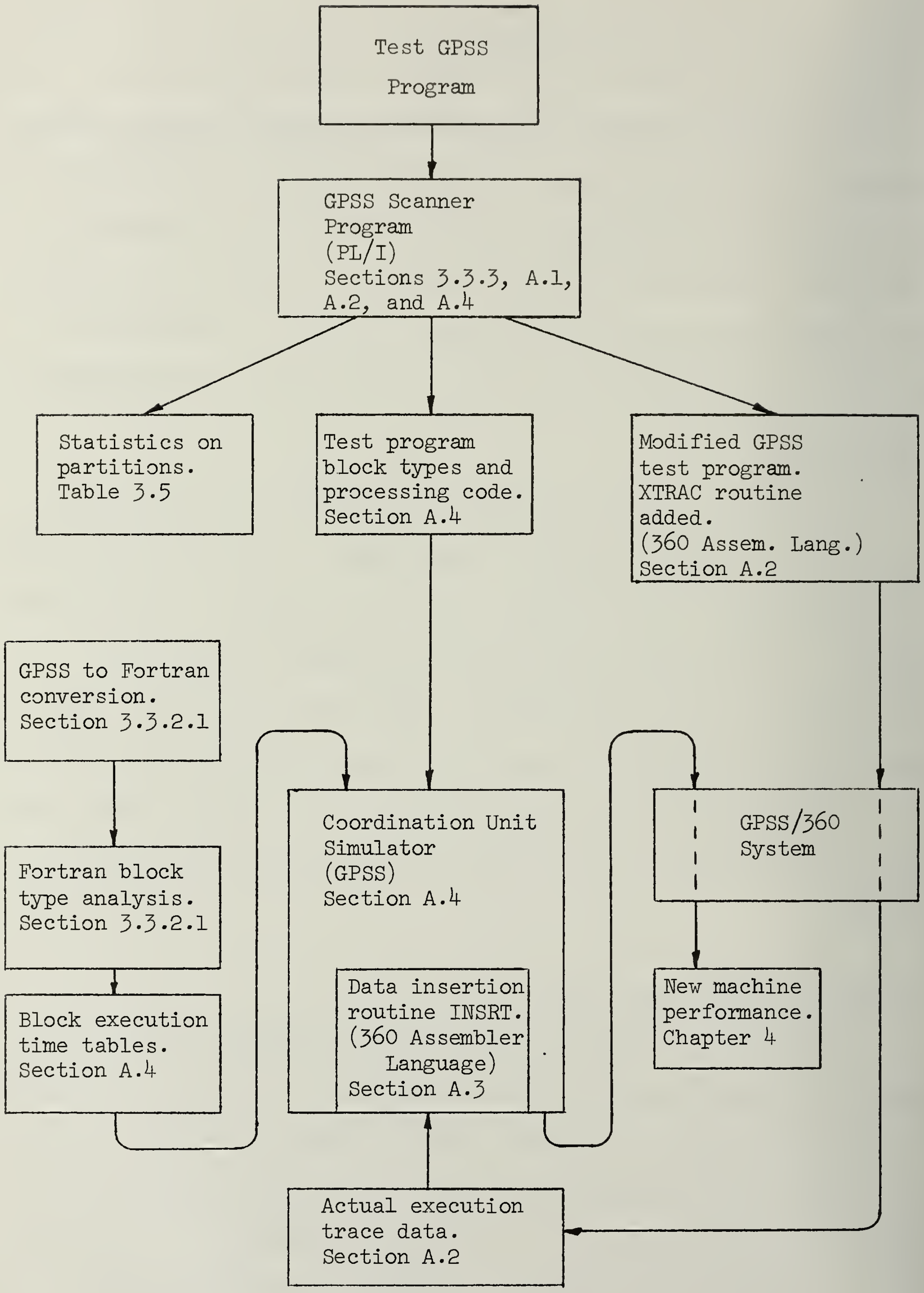

Test GPSS Program
GPSS Scanner Program (PL/I) Sections 3.3.3, A.1, A.2, and A.4
Statistics on partitions. Table 3.5
Test program block types and processing code. Section A.4
Modified GPSS test program. XTRAC routine added. (360 Assem. Lang.) Section A.2
GPSS to Fortran conversion. Section 3.3.2.1
Fortran block type analysis. Section 3.3.2.1
Block execution time tables. Section A.4
Coordination Unit Simulator (GPSS) Section A.4
Data insertion routine INSRT. (360 Assembler Language) Section A.3
GPSS/360 System
New machine performance. Chapter 4
Actual execution trace data. Section A.2

Figure A.1. Simulation Test System

transaction which called for the execution. A third piece of information, the simulated time at which the block is being executed, is needed for this test system.

An assembly language routine was written to gather this data and output it on punched cards. The routine is called by the special GPSS HELP block. The HELP block allows a user to write his own routines to supplement the fixed set provided by GPSS. It was needed here to access the transaction number, a variable not available to the programmer as a standard numerical attribute, and to provide the punched output.

Trace data is not needed at every block in a program. It is needed at those blocks which are the first in a partition of simultaneously executable blocks, or at labeled blocks which may be the first in a run time partition due to a transfer to that block. The scanner program of the previous section generates a card deck containing all the appropriate HELP blocks which are then merged into the original GPSS program. The modified program thus formed is logically equivalent to the original.

Specifically, when a partition ends, the next block must be the first block in the next partition. The scanner program can simply punch a HELP block card following every block for which P, the partition code bit, is zero.

For labeled blocks the action is slightly more involved. For any labeled block it is necessary to give that label to the HELP block, then punch a card for the original block without the label. The original labeled block is replaced by a labeled HELP and an unlabeled copy of itself. If the labeled block is not the first in a partition, any transaction which moves into it from the block directly above does not need the trace data. In this case a TRANSFER block is used to let such a transaction branch around the HELP. Thus three cards are punched.

Note that the addition of these HELP blocks changes the block numbering of the original program when a trace is being taken. For this reason the GPSS internal block number variable cannot be used for the next block data in the trace. The format of the HELP block allows the use of parameters which can be read by the routine. Thus the scanner program, which keeps a block counter, can provide the original block number as a parameter.

The card deck generated by this program contains the essential information on the actual execution of a test program. This deck is input data for the machine simulator. The operation of inserting it into the simulator is done by the program of the next section.

A.3 Trace Data Insertion: INSRT

GPSS is weak in the areas of input and output. It was necessary to write an assembly language routine to load the trace data described in the previous section. The routine is called by a HELP block, as was the data extraction routine.

Savevalue locations in the simulator are reserved for the trace data. A table of 300 half words is used for transaction number data. A table of 300 full words, with the same savevalue index, is used for simulated time data. Within the simulator the same number can be used as the index for a half word savevalue to get a transaction number and as a full word savevalue index to get the corresponding simulated time.

The trace information is completed by the next block number data.

This is stored in a 300 half word table with the index offset by 300 from the transaction number table.

The routine to insert this data can initially load 300 trace records. Trace data is time ordered. Its use is biased towards, but not limited to, the time ordering. Once a record is read it is no longer needed. When data not currently in the tables is needed, the insertion routine will compact the existing unused data and load new records until the tables are full or an end of the data file is reached.

At this point the gathering of data needed for simulating the execution of real GPSS programs and a routine for loading the data into the simulator have been described. The remaining step is the simulation.

A.4 Coordination Unit Simulation

This is a simulator, written in GPSS, of a machine for processing the execution phase of GPSS. The purpose of the simulator is twofold. First, the number of transactions which can be processed concurrently is desired. Multiple transaction concurrency offers parallelism of a type that does not exist in the procedural languages. It cannot be measured by examining the source program. Simulation of the machine in Chapter 4 executing real GPSS programs, using the trace data from actual executions, does provide a measure. The results of this first purpose of the simulator were used in deciding on a reasonable number of task processors. The results are given in Table 4.1.

The second purpose is to determine the capability of the co-ordination unit to select tasks and keep the task processors busy. These results are also given in Table 4.1.

Programming emphasis is on the simulation of the coordination unit. It is assumed that the memory system of the machine does not cause delays which must be simulated. Task processors are simulated only to the extent of requesting a task from the coordination unit, accepting one, and advancing time by the execution time for the task. The task execution time for a block type is the average over all traces of the parallel machine execution time. Analysis of block types for parallelism, covered in section 3.3.2.1, is the source of the time figures. A conversion of 500 nanoseconds per time step, T_p, was used. This allows an average of two or three clock pulses per operation with a 5MHz clock.

Information on the test program whose execution is being simulated is unique to each program and must be supplied to the simulator. Static information consisting of a representation of the program and the processing code is given as a GPSS function. The dynamic trace information is read into the model by the INSRT routine of the previous section.

The function is in the simulator listing with the label TYPCD. There is a function point for each block in the test program. The point gives the block type and processing code for the block. Block types are numbered from one to 44 in alphabetic order. The function definition and follower cards are another part of the scanner program output.

A listing of the simulator follows.

```
// EXEC DUMMY
//DD1 DD DSN=&X,SPACE=(TRK,(20,5,2)),DISP=(,PASS),UNIT=DISK
// EXEC LKEDASM,PARM='LIST,MAP,REUS'
//LKED.SYSLMOD DD DSN=&X(INSRT),DISP=(OLD,PASS)
//LKED.SYSIN DD *
         < OBJECT DECK FOR SUBROUTINE INSRT >
// EXEC PGM=DAG01,PARM='B'
//DINTERO DD UNIT=DISK,SPACE=(CYL,(1,1))
//DINTWORK DD UNIT=DISK,SPACE=(CYL,(1,1))
//DOUTPUT DD SYSOUT=A
//DREPTGEN DD UNIT=DISK,SPACE=(CYL,(1,1))
//DSYMTAB  DD UNIT=DISK,SPACE=(CYL,(1,1))
//STEPLIB DD DSN=&X,DISP=(OLD,PASS)
// DD DSN=SYS1.GPSSLIB,DISP=SHR
//SYSPRINT DD SYSOUT=A
//DINPUT1 DD DDNAME=SYSIN
//SYSIN DD *
       REALLOCATE XAC,200,FSV,600,HSV,800,BLO,300
       SIMULATE
*
* THIS GPSS PROGRAM IS A SIMULATION OF THE CONTROL UNIT OF A MACHINE
* DESIGNED TO RUN SIMULATION PROGRAMS USING CONCURRENT PROCESSING
* TECHNIQUES.
*
* THE BASIC UNIT OF TIME IN THIS SIMULATION IS 100 NANOSECONDS.
*
* TRANSACTION PARAMETER USAGE
*
*         P1  - EXECUTION TIME IN TASK PROCESSORS.
*         P2  - SERIAL NUMBER OF THE XACT BEING SIMULATED.
*         P3  - NUMBER OF NEXT BLOCK IN THE PROGRAM FOR THIS TASK.
*                    VALUE COMES FROM TRACE TABLE FOR MASTER XACTS.
*                    VALUE COMES FROM INCREMENTING MASTER XACT VALUE
*                    FOR SPLIT XACTS.
*         P4  - TRANSACTION SIMULATED TIME AT THE START OF THE CONCUR-
*                    RENCY GROUP. VALUE TAKEN FROM TRACE TABLES.
*         P5  - COUNTER OF TASKS IN A CONCURRENCY GROUP. USED TO
*                    DETERMINE WHEN ALL TASKS IN THE GROUP HAVE BEEN
*                    PROCESSED.
*         P6  - POINTER TO FULLWORD SAVEVALUE IN TIMST WHICH HAS THE
*                    TIME, SIMULATED, FOR THE XACT WITH SERIAL NUMBER
*                    P6-400.    P6=P2+400.
*         P7  - POINTER TO HALFWORD SAVEVALUE IN NOFPE WHICH HAS THE
*                    NUMBER OF PROCESSORS FOR BLOCK NR P7-44. P7=P11+44.
*         P8  - INDICATES TO OUTPUT UNIT THE SOURCE OF THE TASK.
*                    P8 =0 FOR TASKS FROM DELAY QUEUE.
*                    P8 =1 FOR TASKS FROM PROCESS QUEUE.
*         P9  - VALUE OF FUNCTION TYPCD IN FORMAT DBBSP.
*         P10 - CONTAINS BITS DBB OF P9.
*         P11 - NUMERICAL DESIGNATION OF BLOCK TYPE FOR THIS TASK.
*                    BITS BB OF P9
*               POINTER TO HALFWORD SAVEVALUE GIVING EXECUTION TIME FOR
*                    THIS BLOCK TYPE.
*         P12 - BLOCK PROCESSING CODE. BITS SP OF P9.
*         P13 - POINTER TO FULLWORD SAVEVALUE IN TIMST WITH MIN VALUE.
*         P14 - POINTER TO HALFWORD SAVEVALUE WITH SERIAL NUMBER AND TO
*                    FULLWORD SAVEVALUE WITH SIMULATED TIME IN THE
*                    DYNAMIC TRACE TABLES.
*         P15 - POINTER TO HALFWORD SAVEVALUE WITH NEXT BLOCK NUMBER
*                    IN THE TRACE TABLES. P15=P14+300.
*
```

```
* HALFWORD SAVEVALUE TABLES AND DATA
*
*     TABLE OF EXECUTION TIMES FOR GPSS BLOCKS IN THE PARALLEL MACHINE.
*     EXECUTION TIMES ARE TAKEN FROM THE ANALYSIS OF BLOCKS FOR
*     PARALLELISM ON THE SYSTEM REPORTED IN CHAPTER 3. THE CONVERSION
*     FROM T(P) TO TIME IS: ONE STEP IN T(P) =500 NANOSECONDS.
 EXTIM EQU         1(44),H
       INIT        XH1,60         ADVANCE BLOCK TYPICAL EXECUTION TIME
       INIT        XH4,65         ASSIGN
       INIT        XH8,80         DEPART
       INIT        XH9,175        ENTER
       INIT        XH14,110       GENERATE
       INIT        XH16,50        INDEX
       INIT        XH18,150       LEAVE
       INIT        XH19,90        LINK
       INIT        XH20,60        LOGIC
       INIT        XH22,40        MARK
       INIT        XH24,90        MSAVEVALUE
       INIT        XH27,45        PRIORITY
       INIT        XH28,100       QUEUE
       INIT        XH29,70        RELEASE
       INIT        XH32,60        SAVEVALUE
       INIT        XH34,70        SEIZE
       INIT        XH36,180       SPLIT
       INIT        XH38,100       TERMINATE
       INIT        XH39,50        TEST
       INIT        XH41,60        TRANSFER
       INIT        XH42,200       UNLINK
       INIT        XH$DCOD,100    DECODE ROUTINE
*
*     AN ASSEMBLY LANGUAGE ROUTINE, INSRT, IS CALLED BY THE HELP BLOCK.
*     THIS ROUTINE LOADS DATA WHICH REPRESENTS THE EXECUTION TRACE OF
*     TRANSACTIONS IN TEST GPSS PROGRAMS. SIMULATED TRANSACTION SERIAL
*     NUMBERS ARE PUT IN HW SAVEVALUE LOCATIONS 100 TO 299. THE NEXT
*     BLOCK THAT THE TRANSACTION WILL EXECUTE IS LOADED INTO HW
*     SAVEVALUE LOCATIONS 300 TO 599. CORRESPONDING TRANSACTION AND
*     BLOCK DATA ARE THUS OFFSET 300 LOCATIONS. SIMULATED TIME FOR
*     EACH RECORD IS LOADED INTO FW SAVEVALUE LOCATIONS 100 TO 299.
*     THE TRANSACTION NUMBER AND TIME DATA HAVE THE SAME INDICES.
*
*     WHEN A TRACE RECORD IS USED THE TRANSACTION NUMBER FIELD IS
*     SET TO ZERO.  WHEN MORE TRACE RECORDS ARE NEEDED, INSRT WILL
*     COMPACT THE TRACE DATA AND LOAD ADDITIONAL RECORDS UNTIL THE
*     END OF THE TRACE DATA FILE OCCURS.
*
*     POINTER TO LAST VALID ENTRY OF DYNAMIC TRACE DATA.
 LAST  EQU         98,H
       INIT        XH$LAST,99
*     COUNT OF EMPTY LOCATIONS IN TRACE DATA TABLES
 LDCNT EQU         99,H
       INIT        XH$LDCNT,300
*     TEST PROGRAM TRACE DATA. DYNAMIC DATA LOADED BY 'HELP INSRT'.
 XACNR EQU         100(300),H     XACT SERIAL NUMBER
 NXTBK EQU         400(300),H     NEXT BLOCK FOR XACT IN XH(*-300)
       INIT        XH$TSKPR,4     ASSUMES TOTAL OF 4  PROCESSORS
*
* THE FOLLOWING TWO SAVEVALUES DEFINE THE NUMBER OF TRANSACTIONS BEING
* SIMULATED. THEY ARE PARTICULAR TO THE PROGRAM BEING TESTED.
       INIT        XH$SIMX,56     NOF XACTS IN RAIL FLEET II
       INIT        XH$DMSIM,55    NR OF ACTIVE XACTS MINUS ONE
*
```

```
* FULLWORD SAVEVALUE TABLES
*
*     TABLE SXCTR HAS ONE ENTRY PER TRANSACTION. THE ENTRY IS A
*     COUNTER OF TASKS BEING PROCESSED FOR THE TRANSACTION. THE
*     NUMBER OF TRANSACTIONS BEING PROCESSED CONCURRENTLY IS THE
*     NUMBER OF ENTRIES WITH A VALUE GREATER THAN OR EQUAL TO ONE.
*     SINCE THIS TABLE BEGINS AT SAVEVALUE 1, THE TRANSACTION SERIAL
*     NUMBER, P2, CAN BE USED AS THE INDEX.
 SXCTR EQU        1(99),X         USED TO TABULATE SIMULTANEOUS XACTS
*
*     TEST PROGRAM TRACE DATA. SIMULATED TIME OF XACT WITH SERIAL NR
*     GIVEN IN THE HALFWORD SAVEVALUE WITH THE SAME INDEX
 SIMTM EQU        100(300),X
*     SIMULATED TIME OF XACTS BEING SIMULATED
 TIMST EQU        401(100),X      X(400+J) HAS SIM TIME FOR XACT(J)
       INIT       X401-X500,2147483647
       INIT       X$MINTM,1       SIMULATED TIME INITIAL VALUE
       INIT       X$XMAX,2147483647
*
* LOGIC SWITCH USAGE
*
 PARTN EQU        1(100),L        SWITCH(J) FOR XACT(J) ENABLES ASSEMBLY
       INIT       LS1-LS100
*
* FUNCTIONS:
*
*     TYPCD IS PRODUCED AUTOMATICALLY BY THE GPSS ANALYZER PROGRAM. IT
*     IS A LIST OF THE BLOCKS IN THE PROGRAM BEING ANALYZED, THE
*     PROCESSING CODE, AND AN INDICATOR OF BLOCKS WHICH USE FN, V, BV,
*     OR * IN THE OPERAND FIELD AND THUS REQUIRE DECODING.
*     BLOCKS ARE LISTED WITH NUMBERS CORRESPONDING TO ALPHABETIC ORDER
*     BEGINNING WITH 1 FOR ADVANCE AND ENDING WITH 44 FOR WRITE.
*     THE FORMAT FOR THIS PACKED DATA IS DBBSP WHERE:
*        D  IS 1 FOR BLOCKS WHICH REQUIRE OPERAND DECODING; 0 OTHERWISE,
*        BB IS THE NUMERICAL BLOCK TYPE,
*        S IS 0 FOR BLOCKS WHICH USE SYSTEM VARIABLES AND MUST BE
*           DELAYED; 1 OTHERWISE,AND
*        P IS 0 FOR THE LAST BLOCK IN A PARTITION; 1 OTHERWISE.
*
 TYPCD FUNCTION   P3,L049         RAIL FLEET PROGRAM
    1 01411      2 00410      3 03211      4 03601      5 00411      6 00111
    7 03910      8 03810      9 00411     10 04110     11 00411     12 10400
   13 13910     14 10410     15 10400     16 03910     17 03611     18 10101
   19 13901     20 13910     21 02810     22 00900     23 00811     24 01610
   25 00110     26 10111     27 13901     28 13911     29 13910     30 00100
   31 01811     32 03810     33 03810     34 13201     35 04110     36 00110
   37 00111     38 04110     39 00110     40 00111     41 04110     42 01411
   43 00410     44 00111     45 13900     46 13211     47 13900     48 00411
   49 04110
*
*     INTER GIVES THE CUMULATIVE PROBABILITY THAT X INTERROGATIONS
*     ARE REQUIRED IN THE ASSOCIATIVE MEMORY TO DETERMINE THE ENTRY
*     WITH MINIMUM TIME.
 INTER FUNC       RN4,D16
.01,1/.02,2/.04,3/.06,4/.09,5/.13,6/.19,7/.27,8
.38,9/.52,10/.64,11/.74,12/.82,13/.89,14/.95,15/1.00,16
*
 FNDCD FUNC       RN2,D2          FUNCTION EVALUATION REQUIRED
.5,1/1.0,2
*
* THE FIRST PART OF THE PROGRAM LOADS THE FIRST 300 RECORDS OF TRACE
* TABLE DATA AND INITIALIZES CTHER VARIABLES.
        GENE        ,,,1                GENERATE 1 XACT TO LOAD TRACES
        HELP        INSRT               LOAD DYNAMIC TRACE DATA
        QUEUE       REQST,XH$TSKPR      LET ALL PROCESSORS BE AVAILABLE
        SAVEVAL     LSTMN,V9,H          UPPER USED LIMIT ON TIMST
 9      VARIABLE    XH$SIMX+4C0
        TERM        0
*
* CNE TRANSACTION IS GENERATED FOR EVERY TRANSACTION BEING SIMULATED.
        GENE        ,,,XH$SIMX,,17,F        GENERATE 1 XACT PER SIM XACT
        SELECT LS   16,1,100            GET FIRST SET   SWITCH FROM PARTN SET
        LOGIC R     P16                 PREVENT >1 TRACE PER XACT
 START  SELECT E    14,100,XH$LAST,P16,XH,HOLDS
        ASSI        4,X*14              P4 IS GIVEN THE XACT SIMULATED TIME
        ASSI        2,XH*14             P2 IS GIVEN THE XACT SERIAL NR
        SAVEVAL     *14,K0,H            RESET THE XACT NR SAVEVAL. DATA USED.
        ASSI        6,V1                P6 POINTS TO X(J) IN TIMST
 1      VARI        P2+400              TIMST SAVEVALUES BEGIN AT X401
        SAVEVAL     P6,P4               X(J) GETS SIM TIME FOR XACT(J-400)
        ASSI        15,V2               P15 IS PCINTER TO NEXT BLOCK XH
 2      VARI        P14+300             NEXT BLOCK XH OFFSET 300 FROM XACT NR
        ASSI        3,XH*15             P3 IS GIVEN THE XACT NEXT BLOCK
*
* ASSOCIATIVE MEMORY: SELECT THE XACT WITH MINIMUM SIMULATED TIME AS
*                     THE BEST CANDICATE FCR PROCESSING
 AMEM   ENTER       AMEM                ENTER AM
        GATE LR     RDWRT               ALLOW LOAD IF AM NOT BUSY
        LOGIC S     RDWRT               SET SWITCH TO INDICATE BUSY
        ADVA        2                   TIME TO LOAD AM
        LOGIC R     RDWRT               RESET BUSY SWITCH
        JOIN        AVAIL,P2            XACT IS AVAILABLE FOR SELECTION
        LOGIC S     CTL                 SWITCH SET WHEN XACTS ARE ON AMEM
        LINK        AMEM,P4             CRDERED CHAIN TO SIMULATE AM
*
* THE TRANSACTION GENERATED NEXT SIMULATES INTERROGATION OF THE
* ASSOCIATIVE MEMORY.
        GENE        ,,,1,,1             XACT TO INTERROGATE AM
 INT    GATE LR     RDWRT               ALLOW INTERROGATION IF AM NOT BUSY
        LOGIC S     RDWRT               SET SWITCH TO INDICATE BUSY
        ADVA        1,FN$INTER          INTERROGATION TIME
        LOGIC R     RDWRT               RESET BUSY SWITCH
        UNLINK      AMEM,AAA,K1,,,XXX SELECT MIN TIME XACT
        TEST E      XH$XCTR,K0          WAIT FOR UNLINKED XACTS TO BE MOVED
        TRANS       ,INT                RESUME INTERROGATION
 XXX    LOGIC R     CTL                 SWITCH TO CONTROL INTERROGATION
        GATE LS     CTL                 WAIT FOR XACTS TO GET ON CHAIN
        TRANS       ,INT
 AAA    UNLINK      AMEM,QTEST,ALL,4 UNLINK ALL XACTS WITH SAME SIM TIME
 QTEST  SAVEVAL     XCTR+,K1,H          CCUNT UNLINKED XACTS
        ADVA        1                   RESOLVE EACH RESPONDER
        EXAM        DELQ,P2,RCUTE       IS XACT CN DELAY QUEUE?
        TEST LE     P4,X$MINTM,NCTMN YES. IS IT MIN TIME?
        REMCVE      AVAIL,,P2           YES. XACT HAS BEEN SELECTED
        SAVEVAL     XCTR-,K1,H          DECREMENT CTR AS XACTS ARE HANDLED
        LEAVE       AMEM
        TRANS       ,LVDLY              XACT IS LEAVING DELAY QUEUE
 NOTMN  SAVEVAL     XCTR-,K1,H          DECREMENT CTR AS XACTS ARE HANDLED
        LINK        TEMPL,LIFC          XACT CANNOT BE SELECTED
 RELNK  LINK        AMEM,LIFO
*
* CODE EVALUATION UNIT: ROUTES TASKS TO PROPER QUEUE AND SWITCHES
*                         TRANSACTIONS AT PARTITION BOUNDARIES
*
 ROUTE SAVEVAL     XCTR-,K1,H           DECREMENT CTR AS XACTS ARE HANDLED
       SEIZE       CODE                 GET CODE EVALUATION UNIT
       REMOVE      AVAIL,,P2            XACT HAS BEEN SELECTED
       LEAVE       AMEM
       TEST E      P4,X$MINTM,PAC       IS THIS A MIN TIME XACT?
       LOGIC S     BYPAS                YES. ENABLE DELAY QUEUE BYPASS
 PAC   ASSI        9,FN$TYPCD           P9 GETS BLOCK TYPE AND CODE
       ADVA        1                    EVALUATE EACH PROCESSING CODE
       ASSI        10,V3                P10 >99 MEANS BLOCK OPERANDS NEED
 3     VARI        P9/100                 DECODED, INCREASING EXECUTION TIME
       ASSI        11,V4                P11 GETS BLOCK TYPE. NUMBERS 1 THRU
 4     VARI        P10@100                44 = BLOCKS ADVANCE THRU WRITE
       ASSI        12,V5                P12 GETS THE 2 BIT BLOCK CODE
 5     VARI        P9@100
       ASSI        7,V6                 P7  POINTS TO THE XH GIVING THE NR OF
 6     VARI        P11+44                 PROCESSORS BLOCK P11 USES
       ASSI        5+,K1                INCR COUNT OF TASKS IN CONCURRENCY GRP
       TEST E      P12,K11,PAS,1        DOES CODE SAY PROCESS Q AND CONTINUE?
       GATE LR     DELAY,PDAC           YES. IS DELAY SWITCH RESET?
 BDPAC SPLIT       K1,PRQA              YES. SEND A TASK TO THE PROCESS Q.
       ASSI        3+,K1                INCR BLOCK POINTER
       TRANS       ,PAC                 TRY THE SAME XACT ON THE NEXT BLOCK
 PAS   TEST E      P12,K10,DAC,1        DOES CODE SAY PROCESS Q AND SWITCH?
       GATE LR     DELAY,PDAS           YES. IS DELAY SWITCH RESET?
 BDPAS RELE        CODE                 YES. RELEASE CODE TO SWITCH XACTS
       LOGIC R     BYPAS                RESET DELAY QUEUE BYPASS
       TRANS       ,PRQB                SEND THE MASTER TASK TO THE PROCESS Q
 DAC   TEST E      P12,K01,DAS          DOES CODE SAY DELAY Q AND CONTINUE?
       GATE LR     BYPAS,BDPAC          YES. SHOULD THIS XACT BYPASS DELAY Q?
       LOGIC S     DELAY                NO.  SET THE DELAY SWITCH
 PDAC  SPLIT       K1,DLQA              SEND THE TASK TO THE DELAY Q
       ASSI        3+,K1                INCR BLOCK POINTER
       TRANS       ,PAC                 TRY SAME XACT ON THE NEXT BLOCK
 DAS   TEST E      P12,K00,ERRCD        DOES CODE SAY DELAY Q AND SWITCH?
       GATE LR     BYPAS,BDPAS          YES. SHOULD THIS XACT BYPASS DELAY Q?
 PDAS  LOGIC R     DELAY                NO.  RESET THE DELAY SWITCH
       RELE        CODE                 PREPARE TO SWITCH XACTS
       QUEUE       DELAY
       JOIN        DELQ,P2              GROUP DELAYED XACTS
       ASSI        8,K0                 P8 =0 INDICATES DELAY QUEUE IS SOURCE
       TRANS       ,AMEM                DELAYED XACTS REMAIN IN THE AM
*
* DELAY QUEUE: WHEN THE SIMULATED TIME OF A XACT IN THIS QUEUE IS EQUAL
*              TO THE MIN TIME OF ALL XACTS, ALL TASKS IN THE PARTITION
*              ARE ELIGIBLE TO LEAVE THE QUEUE. TASKS FROM THIS
*              QUEUE HAVE HIGHER PRIORITY FOR USE OF THE PROCESSORS.
*
 DLQA  ASSI        5,K0                 RESET PARTITION TASK COUNTER.
       QUEUE       DELAY
       ADVA        2                    TIME TO LOAD THE QUEUE
       JOIN        DELQ,P2              GROUP DELAYED XACTS
       ASSI        8,K0                 P8 =0 INDICATES DELAY Q IS SOURCE
       TEST E      G$AVAIL,K0,MERGE,1   IS AVAILABLE GROUP EMPTY?
       GATE SE     PRPUL,MERGE,,1       YES. ARE PROCESSORS IDLE?
       TRANS       ,TASK                YES. PROCESS THIS XACT
 MERGE LINK        DELAY,P4             MERGE BY TIME.
 LVDLY REMOVE      DELQ,,P2             XACT IS LEAVING THE DELAY Q
       ADVA        2                    REMOVE TASKS FROM DELAY QUEUE
       UNLINK      DELAY,LNKCT,ALL,2    UNLINK ALL TASKS FOR THIS XACT
```

```
LNKOT LINK        OUTPT,P8,TASK
*
* PROCESS QUEUE:  TASKS ARE AVAILABLE FOR PROCESSING UPON REQUEST
*
                                     RESET CONCURRENCY GROUP TASK COUNTER
 PRQA  ASSI       5,K0
 PROB  QUEUE      PROCS
       ADVA       2                  TIME TO LOAD THE QUEUE
       ASSI       8,K1               P8 =1 WHEN PROCESS Q IS THE SOURCE
       LINK       OUTPT,P8,TASK      P8 IS PRIORITY FOR OUTPUT UNIT
*
* OUTPUT UNIT: THIS UNIT SELECTS THE NEXT TASK TO BE SENT TO THE
*              PROCESSORS. TASKS FROM THE DELAY QUEUE ARE GIVEN PRIORITY
*
 DLYQ  DEPART     DELAY              THE DELAY Q IS OUTPUTTING
       TRANS      ,DEQ
 TASK  SEIZE      OUTPT              PLACE TASK IN OUTPUT UNIT
       ADVA       2                  MOVE TASK THROUGH OUTPUT UNIT
       TEST E     P8,K1,DLYQ         DID THE TASK COME FROM THE PROCESS Q?
       DEPART     PROCS              YES.
 DEQ   TEST G     Q$REQST,K0         DOES A TASK REQUEST EXIST?
       UNLINK     OUTPT,TASK,K1      REMOVE NEXT TASK FROM Q
*
* TASK PROCESSORS: EACH TASK USES A PROCESSOR FOR THE TIME DETERMINED
*                  BY THE BLOCK TYPE AND OPERANDS.
*
*                                    THE TASK USES PROCESSORS FROM A POOL
       ENTER      PRPUL              YES. MOVE FROM OUTPUT TO PROCESSORS
       RELEASE    OUTPT              REMOVE THE TASK REQUEST
       DEPART     REQST              COUNT TASKS IN PROCESS FOR THIS XACT
       SAVEVAL    P2+,K1             P11 IS THE INDEX FOR TABLE EXTIM
       ASSI       1,XH*11            WILL JUST THE BLOCK EXECUTE TIME DO?
       TEST L     P10,K99,MRTIM      YES. P1 HAS TOTAL EXECUTION TIME
 TIME  ADVA       P1
       TABULATE   TIMEX
       SAVEVAL    P2-,K1             COUNT TASKS IN PROCESS FOR THIS XACT
       LEAVE      PRPUL              PROCESSORS BECOME AVAILABLE AGAIN
       QUEUE      REQST              REQUEST A TASK
       TEST E     P5,K0,MASTR,1      IS THIS THE MASTER XACT?
       GATE LS    P2                 NO. HAS THE MASTER TASK BEEN PROCESSED
 ASSEM ASSEM      P5                 YES. ASSEMBLE TASKS IN CONCURRENCY GP
       LOGIC R    P2                 CLOSE GATE FOR NEXT ITERATION
       TABULATE   PART
       TEST NE    P11,K38,TERMB      IS THE BLOCK TYPE OTHER THAN TERMINATE
       ASSI       5,K0               YES. RE-INITIALIZE THE CON GP COUNTER
 CONT  SELECT E   14,K100,XH$LAST,P2,XH,HOLDC   GET NEXT DATA FOR XACT
       ASSI       4,X*14             UPDATED SIM TIME FOR THIS XACT
       SAVEVAL    *14,K0,H           RESET THE XACT NR SAVEVAL. DATA USED.
       SAVEVAL    EMPTY+,K1,H        COUNT EMPTY TRACE TABLE LOCATIONS
       SAVEVAL    P6,P4              UPDATE TABLE OF CURRENT SIM TIMES
       ASSI       15,V2              P15 IS POINTER TO NEXT BLOCK XH
       ASSI       3,XH*15            P3 IS GIVEN THE XACT NEXT BLOCK
       TEST L     XH$EMPTY,K50,LDATA ARE THERE LESS THAN 50 EMPTIES?
 GTMIN SELECT MIN 13,401,XH$LSTMN,,X P13 WILL POINT TO MIN TIME SAVEV
       TEST G     X*13,X$MINTM,AMEM HAS MIN TIME CHANGED?
       SAVEVAL    MINTM,X*13         YES. UPDATE IT
       UNLINK     TEMPL,RELNK,ALL   UNLINK DELAY XACTS FROM TEMP CHAIN
       TRANS      ,AMEM
 TERMB TEST G     XH$DMSIM,K0,TRM     ARE THERE OTHER XACTS?
       TEST G     XH$DMSIM,V7,UNHLD YES. ARE REMAINING XACTS IN HOLD?
 7     VARIABLE   CH$HOLDS+CH$HOLDC SUM OF XACTS WAITING FOR TRACE DATA
       SAVEVAL    P6,X$XMAX           NO. SET SIM TIME TO MAX AND TERM
       SAVEVAL    DMSIM-,K1,H         DECREMENT COUNT OF ACTIVE SIM XACTS
```

```
      SELECT MIN 13,401,XH$LSTMN,,X P13 WILL POINT TO MIN TIME SAVEV
      TEST G    X*13,X$MINTM,TRM  HAS MIN TIME CHANGED?
      SAVEVAL   MINTM,X*13        YES. UPDATE IT
      UNLINK    TEMPL,RELNK,ALL,,,TRM  UNLINK DELAY XACTS FROM TEMPL
      LOGIC S   CTL
 TRM  TERM      1
 MASTR LOGIC S  P2                OPEN GATE FOR TASKS IN CONCURRRENCY GP
      TRANS     ,ASSEM
 MRTIM ASSI     1+,XH$DCOD,3      THE BLOCK USES FN, *, V, OR BV
      TRANS     ,TIME
 LDATA SAVEVAL  EMPTY,K0,H        RESET EMPTIES COUNTER
      TEST G    V7,K0,GTMIN       ARE XACTS WAITING FOR TRACE DATA?
      HELP      INSRT             YES. LOAD DATA
      ASSI      1,125             IDENTIFY OCCURRENCE OF INSRT
      PRINT     ,,MOV,X           IDENTIFY OCCURRENCE OF INSRT
      TEST NE   XH$LDCNT,K499,GTMIN WAS INSERT GOOD
      UNLINK    HOLDS,START,ALL YES. ATTEMPT TO START HOLDS XACTS
      UNLINK    HOLDC,CONT,ALL    ATTEMPT TO CONTINUE HOLDC XACTS
      TRANS     ,GTMIN
 HOLDC TEST NE  XH$LDCNT,K999,TERMB   IS THERE MORE TRACE DATA?
      LINK      HOLDC,FIFO        YES. HOLD XACT FOR NEXT INSRT CALL
 HOLDS TEST NE  XH$LDCNT,K999,DLETE   IS THERE MORE TRACE DATA?
      LINK      HOLDS,FIFO        YES. HOLD XACT FOR NEXT INSRT CALL
 UNHLD HELP     INSRT             LOAD TRACE DATA
      ASSI      1,152             IDENTIFY OCCURRENCE OF INSRT
      PRINT     ,,MOV,X           IDENTIFY OCCURRENCE OF INSRT
      TEST NE   XH$LDCNT,K499,ALL WAS INSERT SUCCESSFUL?
      UNLINK    HOLDS,START,ALL YES. ATTEMPT TO START HOLDS XACTS
      UNLINK    HOLDC,CONT,ALL    ATTEMPT TO CONTINUE HOLDC XACTS
      ACVA      1                 ALLOW UNLINKING
      TEST G    XH$DMSIM,V7,ALL WAS THE INSRT OK?
      TRANS     ,TERMB            YES.
 ALL  TERM      XH$SIMX           STOP THE SIMULATION. TRACE DATA FAILURE.
 DLETE SAVEVAL  DMSIM-,K1,H       DECREMENT COUNT OF ACTIVE SIM XACTS
      TERM      1
 ERRCD ASSI     12,K0             BLOCK CODE HAD ERROR. SET CODE TO 00
      TRANS     ,PDAS             RESUME WITH WORST CASE CODE
      GENE      40,,,,,2          GATHER STATISTICS AT 40 TIME UNIT INT
      COUNT GE  2,1,XH$SIMX,1,X  COUNT NR OF SIMULTANEOUS XACTS
      TABULATE  NRSX1
      TABULATE  PRUSE
      TERM      0
 PRUSE TABLE    S$PRPUL,0,1,120
 NRSX1 TABLE    P2,0,1,50         NUMBER OF SIMULTANEOUS XACTS IN PRS
 PART TABLE     P5,1,1,20         RUN TIME PARTITION LENGTH
 TIMEX TABLE    P1,40,10,50       EXECUTION TIME PER BLOCK
 PRPUL STORAGE  4
      START     56                NUMBER OF SIMULATED XACTS
      END
//INDATA DD *
       < TRACE DATA READ BY ROUTINE INSRT >
```

LIST OF REFERENCES

[1] ACM Profession Development Seminar, "Simulation of Discrete Systems," ACM, pp. 123-135.

[2] C. Cartegini, "Scanner for the Analysis of Parallelism in Fortran Programs and IF-Tree Detection," (M.S. Thesis) University of Illinois at Urbana-Champaign, Department of Computer Science; 1971.

[3] Control Data Corporation, "The STAR Computing System." A technical proposal to The Atomic Energy Commission. December 1966.

[4] O. Dahl, and K. Nygaard, "SIMULA: An Algol-Based Simulation Language," Communications of the ACM, p. 671; September 1966.

[5] L. C. Fulmer, and W. C. Meilander, "A Modular Plated Wire Associative Processor," Proceedings of the IEEE Computer Group Conference, pp. 325-335; June 1970.

[6] International Business Machines Corporation, "Capital Investment Studies Using GPSS: Bulk Material Movement Problems," First Edition, p. 39; 1968.

[7] International Business Machines Corporation, "General Purpose Simulation System/360 User's Manual," Fourth Edition; January 1970.

[8] P. J. Kiviat, R. Villanueva, and H. M. Markowitz, The SIMSCRIPT II Programming Language, Prentice-Hall, Inc.; 1968.

[9] D. J. Kuck, "ILLIAC IV Software and Application Programming," IEEE Transactions on Computers, Vol. C-17, No. 8, pp. 758-770; August 1968.

[10] D. J. Kuck, Y. Muraoka, and S. C. Chen, "On the Number of Operations Simultaneously Executable in FORTRAN-Like Programs and Their Resulting Speed-Up," to be published in IEEE Transactions on Computers.

[11] S. E. McAulay, "Job Stream Simulation Using a Channel Multiprogramming Feature," Fourth Conference on Applications of Simulation, ACM, pp. 190-194; 1970.

[12] T. B. Pinkerton, "Program Behavior and Control in Virtual Storage Computer Systems," (Ph.D. Thesis) The University of Michigan, CONCOMP Technical Report 4; April 1968.

[13] Paul F. Roth, "The BOSS Simulator - An Introduction," Fourth Conference on Applications of Simulation, ACM, pp. 244-250; 1970.

[14] R. A. Schwarz, and T. J. Schriber, "Application of GPSS/360 to Job Shop Scheduling," Digest of the Second Conference on Applications of Simulation, ACM, pp. 237-248; 1968.

[15] D. L. Slotnick, et. al., "The ILLIAC IV Computer," IEEE Transactions on Computers, Vol. C-17, No. 8, pp. 746-757; August 1968.

[16] D. G. Weamer, "QUICKSIM - A Block Structured Simulation Language Written in SIMSCRIPT," Third Conference on Applications of Simulation, ACM, pp. 1-11; 1969.

VITA

Edward Willmore Davis, Jr. was born in Akron, Ohio, in 1941. He graduated from The University of Akron in 1964 with a Bachelor of Science in Electrical Engineering degree and earned the Master of Science in Engineering degree there in 1967.

From 1964 to 1968 he was employed in the Computer Engineering Department of Goodyear Aerospace Corporation, Akron, Ohio. In 1968 he entered the University of Illinois Department of Computer Science. He was a research assistant with the Illiac IV Project from 1968 to 1970 and with the Center for Advanced Computation in 1970 and 1971. In 1971 he joined a group studying computer organization and software, where he did research on concurrent processing systems.

GRAPHIC DATA	1. Report No. UIUCDCS-R-72-527	2.	3. Recipient's Accession No.
and Subtitle ULTIPROCESSOR FOR SIMULATION APPLICATIONS			5. Report Date June 1972
			6.
or(s) ard Willmore Davis, Jr.			8. Performing Organization Rept. No. UIUCDCS-R-72-527
rming Organization Name and Address versity of Illinois at Urbana-Champaign artment of Computer Science ana, Illinois 61801			10. Project/Task/Work Unit No.
			11. Contract/Grant No. US NSF GJ 27446
nsoring Organization Name and Address ional Science Foundation hington, D.C.			13. Type of Report & Period Covered Doctoral - 1972
			14.

plementary Notes

tracts

Multiprocessor systems have generally been designed for applications with arrays data which can be operated on in parallel. In this paper an application area ch does not contain such readily identifiable parallelism is examined. Discrete e simulation is found to contain several distinct levels at which potential for current execution exists. The levels are used to guide the organization of a tiprocessor designed for simulation applications.

Both software and hardware aspects of the problem are covered. Features of the tem include a special processor used to evaluate conditional jump trees; clusters simple, fixed point arithmetic processors; a unit to form and dispatch tasks to processors; and a memory system which includes a read only program memory.

Words and Document Analysis. 17a. Descriptors

cial Purpose Computer
ulation Processor
allel Computation

ntifiers/Open-Ended Terms

SATI Field/Group

ilability Statement RELEASE UNLIMITED	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 161
	20. Security Class (This Page UNCLASSIFIED	22. Price

TIS-35 (10-70) USCOMM-DC 40329-P71